NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

VISUALIZATION OF IMPROVED TARGET ACQUISITION ALGORITHM FOR JANUS (A)

by

Mark R. Whitney

December 1994

Thesis Advisor: Morris R. Driels

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE VISUALIZATION OF IMPROVED TARGET ACQUISITION ALGORITHM FOR JANUS (A)		5. FUNDING NUMBERS
6. AUTHOR(S) Mark R. Whitney		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This thesis successfully demonstrates the ability to apply realistic visualization to a training application with a commercially available software program. It is increasingly important with the trend of decreasing military spending to maintain force readiness through quality training. Utilization of realistic visualization in simulations of realtime scenarios is essential to achieving this quality training. This thesis utilizes the Improved Target Acquisition Algorithm for Janus (A) in the visualization of sensor to moving target lines of sight. This visualization takes place in an obstruction constrained terrain using a representation of the higher resolution one meter style Pegasus database numbers.

14. SUBJECT TERMS Pegasus, Janus (A), WTK, and Line Of Sight.			15. NUMBER OF PAGES 171
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

VISUALIZATION OF IMPROVED TARGET ACQUISITION ALGORITHM FOR JANUS (A)

by

Mark R. Whitney
Lieutenant Commander, United States Navy
B.S., Maine Maritime Academy, 1984

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1994

Author: [illegible]
Mark R. Whitney

Approved by: [illegible]
Morris R. Driels, Thesis Advisor

[illegible]
Matthew D. Kelleher, Chairman
Department of Mechanical Engineering

ABSTRACT

This thesis successfully demonstrates the ability to apply realistic visualization to a training application with a commercially available software program. It is increasingly important with the trend of decreasing military spending to maintain force readiness through quality training. Utilization of realistic visualization in simulations of realtime scenarios is essential to achieving this quality training. This thesis utilizes the Improved Target Acquisition Algorithm for Janus (A) in the visualization of sensor to moving target lines of sight. This visualization takes place in an obstruction constrained terrain using a representation of the higher resolution one meter style Pegasus database numbers.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

The author would like to acknowledge the financial support of TRADOC Analysis Command (TRAC) Monterey (Contract No. RCXLD and RCX4D) for the work reported in this thesis.

I. BACKGROUND

A. INTRODUCTION

Operating at sea, out in the field, or in the air is how military personnel receive a majority of their training in order to become proficient at their jobs. By practicing with actual equipment and situations, personnel are able to absorb critical information in order to make intelligent informed decisions. With decreasing military budgets, operational training has had to be severely curtailed due to lack of funds. To maintain operational readiness, today's military has to rely heavily on "schoolhouse" training instead of "operational" training. There has to be a way to train in a schoolhouse environment and yet achieve an operational environment as close to actually being out there in order to maintain proficiency in critical skills.

This is where simulation becomes an integral part of training. In order for simulation to achieve its goal of realistic training, it must be sensed as real. Visualization gives simulation this sense. In a simulation, it is critical to be able to see what looks like a ship, tank, or plane moving across a real terrain in real time, instead of an object moving across a two dimensional contour plane.

Having the capability to train with realistic visualization and do it cost efficiently is a top priority. Cost effectiveness is based in part on the ability to not have to reinvent the wheel for every different type of simulation. If software packages available commercially can be adapted into existing simulation programs, with minimal programming, cost effectiveness would be achieved. This is the major thrust of this thesis.

B. JANUS

1. Background

Janus Army or Janus (A) is a computer based, interactive, two-sided, closed, stochastic, ground combat simulation. Janus simulates two opposing forces in battle. Typically the largest force used is a brigade due to effects on the simulation and ability of force players to command their subordinate forces. These forces are controlled by two or more users operating from individual computer terminals. Each terminal is a graphical depiction of friendly forces, detected enemy forces, terrain contours, and a variety of other useful information. Within Janus, there are thirteen executable FORTRAN programs. These programs are divided into three major groups: database creation and management; scenario creation, verification, and operation; scenario analysis. [Ref. 1: pp. 1-5]

2. Areas of Interest

In order to understand the basics of the improved target acquisition algorithm, it is first necessary to comprehend two major concepts of Janus. First, how targets are acquired. Second, how a Line Of Sight (LOS) is determined.

a. Target Acquisition

The following is the summarization of A.D. Kellner's 14 July, 1992 memorandum for the record on target acquisition in Janus presented in Lieutenant Frederick W. Dau's thesis [Ref. 2: pp. 3-7].

(1) Background. Acquisition in the Janus model is based on the mathematical detection model developed by the Night Vision and Electro-Optics Laboratory (NVEOL). This model is based on a concept involving the computation, for a specific sensor-target pair, of the number of resolved cycles across a target's critical dimension.

Imagine a pattern of stripes or bars equal in length and alternating in color at the targets location. Let the contrast between the colors of the pattern be the same as the contrast between the image of the target and it's background. The length of the pattern is the same as the target's minimum presented dimension. Slowly decrease the width of the strips until the minimum width at which the observer can still distinguish the individual stripes is reached. The number of pairs now contained in the minimum presented area is called the number of resolved cycles for that target-sensor combination.

The concept of resolvable cycles is very useful in the computation of detection probabilities. The NVEOL model defines two probabilities associated with the detection of a given target by a given sensor. First, the Probability of Detection (PD) is that the target will be detected by the sensor given infinite time. This quantity is also known as p-infinity. Second, is the probability that the target will be detected during the time it is within the sensors's field of view, given that it can eventually be detected. This is called P(t). Both quantities of PD and P(t) are functions of the number of resolvable cycles, N, which can be resolved by a given sensor-target pair. The probability that a sensor can discriminate the target once detected is also a function of N.

The portion of the NVEOL model relevant to Janus consist of the following three steps: 1) calculate the attenuation of the target's signature along the LOS 2) given the target's signature at the sensor, calculate the number of cycles the sensor can resolve across the target's critical dimension 3) given N, determine if the target can be detected, acquired, and recognized.

(2) Signature Attenuation. Signature attenuation between the target and the sensor is caused by atmospheric effects as well as objects obscuring the path of the LOS which will be called large area smoke. Let:

St = signature at target

Ss = signature at sensor

T1 = transmission of normal atmosphere

T2 = transmission of large area smoke

and

$$Ss = St * T1 * T2 \tag{1}$$

For optical sensors, St is the optical contrast of the target which is a part of the master database, and thus remains the same for a target during a Janus run.

(3) Resolvable Cycles. The performance of a sensor is represented by a curve of resolvable Cycles Per Milliradian (CMR) as a function of the target signature measured at the sensor. This can be expressed mathematically as follows:

$$CMR = f(Ss) \tag{2}$$

However there is no analytical equation to describe this function, so this function is input in the master database for each sensor as a table of values. Entering the table of the specific sensor-target pair with the signature at the sensor, Ss, produces an output of CMR. Once the number of resolvable cycles is obtained the number of cycles actually resolved by the sensor is obtained by:

$$N = CMR * \frac{TDIM}{Range} \tag{3}$$

where

TDIM = target's minimum presented dimension in meters

Range = sensor to target range in kilometers

TDIM is obtained from the master database for the particular target and range is provided by the simulation.

(4) Acquisition. After the number of cycles resolved on the target is known the probability of detection and time till detection, P(t), may be determined. The probability of detection given by:

$$PD = \frac{CR^{W}}{1 + CR^{W}} \tag{4}$$

where

$W = 2.7 + 0.7^{CR}$

CR = N/N50

N = number of resolvable cycles

N50 = median number of resolvable cycles required for eventual detection

Note that when N = N50 the equation 4 goes to 0.5, meaning that for a random sample of observers, half will require less than N50 resolvable cycles to detect the target and half will require more. Janus uses the following N50 criteria for detection and subsequent target discrimination:

1.0 cycles Detection:	sensing a foreign object is present
2.0 cycles Aimpoint:	ability to aim at a target
3.5 cycles Recognition:	ability to distinguish target class, such as tank, truck, jeep
6.4 cycles Identification:	ability to classify target as specific member of a class

At the beginning of a Janus run each sensor-target pair is assigned a random value which represents the ability of the sensor to detect the corresponding target. This random number is the number of resolved cycles required to detect and is called the detection threshold. When N is greater than or equal to the detection threshold it is said that the p-infinity test has been passed and that the target may eventually be detected by that particular sensor. Once the p-infinity test is passed a value for P(t) is determined using the values of CR and PD. This value is compared to another randomly drawn number, and if it exceeds this random number then detection or another level of discrimination is achieved.

b. Line of Sight

Within the DOLOS subroutine, target acquisition cannot occur if targets are sufficiently concealed behind terrain features. It is therefore important to verify that a LOS between the sensor and target exists before considering target acquisition. Within Janus, the subroutine DOLOS is called to determine Probability of Line Of Sight (PLOS). Initially DOLOS retrieves sensor and target grid locations and elevations and determines a Temporary Probability of Line Of Sight (TPLOS). [Ref. 3: p. 413]

Within the Janus terrain database, each grid is assigned a density value. Assignment of a density value to a grid is determined by surveying the number and height of vegetation for each grid. Once a density value is signed, the entire grid will have the same height and PLOS. [Ref. 2: p. 8]

The slope of a line between sensor and target is now determined based on x and y coordinates of each. Depending on slope, DOLOS incrementally steps one grid at a time starting from sensor and goes toward target. Vertical slope is also calculated. At each step, height changes are added to sensor height. In each grid along the path of the LOS, HITREE retrieves grid tree height and density values. Tree height is added to a grid's ground elevation and compared to LOS elevation in grid. Termination would occur if LOS height is less than grids base elevation. Continuation of LOS would occur if LOS height is greater than the grids base elevation, and procedure is repeated. If LOS height falls somewhere in between grid elevation and total elevation including obstacles, TPLOS is multiplied by the probability of LOS retrieved by HITREE. This repetitive procedure is continued until target grid is reached or TPLOS falls below 0.01. This is an indication that the LOS has been completely attenuated by the terrain features along the path and no longer exists. When DOLOS completes its run, it passes final value of TPLOS back to calling routine as PLOS. [Ref. 2: pp. 9-10]

3. Limitations

a. Terrain Representation

Within the Janus database, the low resolution of terrain and objects on it are considered a limitation to the realism of the simulation. For the purpose of this thesis only terrain, trees, and buildings will be discussed.

The terrain used in Janus is a digitized representation based on Defense Mapping Agency data. As shown in Figure 1 [Ref. 4: p. 6], the graphical representation details contour lines, roads, rivers, vegetation, and other obstructions [Ref 5: p. 3].

The horizontal resolution is one hundred meters and vertical resolution is ten meters. There are several different sizes of exercise areas to choose from at the beginning of a simulation. The area chosen is subdivided into grid cells. These grid cells are represented as a flat plate with constant elevation, with constant terrain type. In order to go between adjacent cells of different elevation it is necessary to "step up" instead of a gradual slope. For LOS calculation it is necessary to linearly interpolate between cells. The resolution of this database is considered to be low, and it does not actually represent actual objects on the terrain. Vegetation is only modeled as an impediment to LOS calculations. Individual bushes or trees are not modeled, and any vegetation shown is constant for each grid cell. Buildings are represented only as statistical entities, individual buildings cannot be shown.

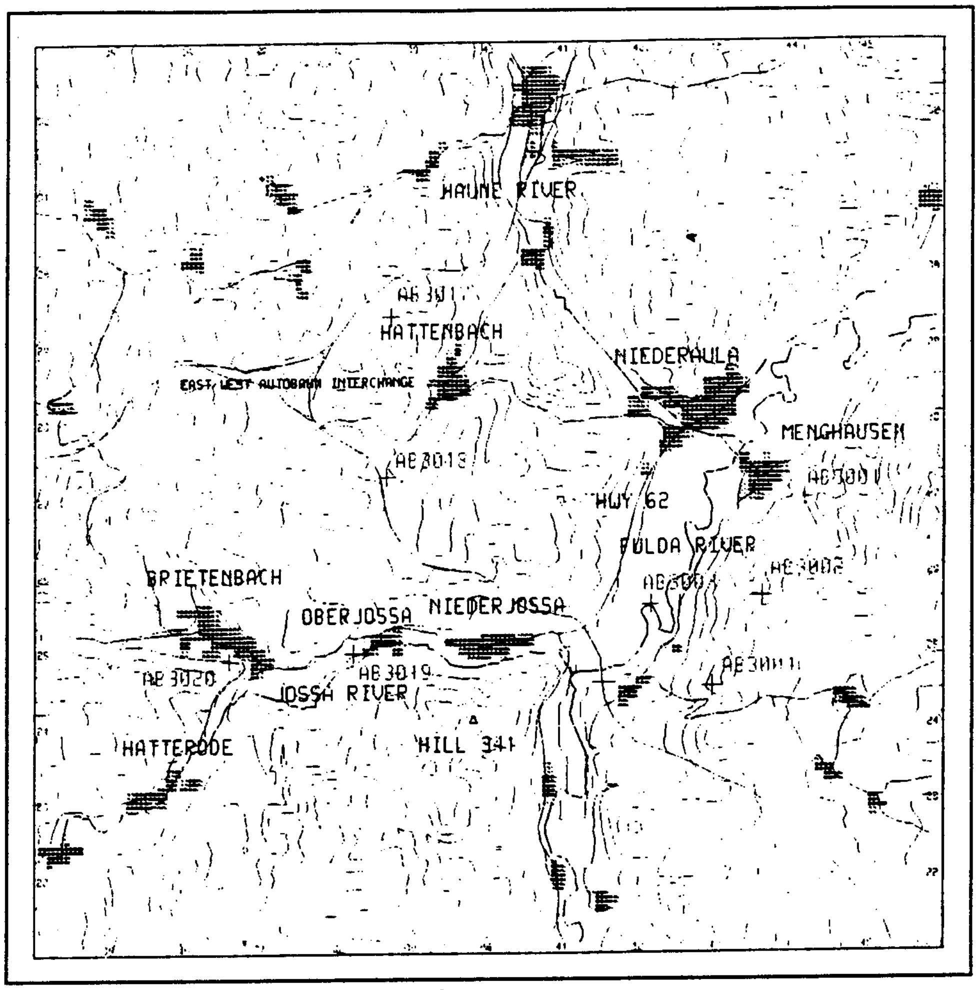


Figure 1: Terrain Representation

b. Target Representation

The view presented by the computer are contour maps of the battlefield consisting of player's subordinate combat systems, units, and personnel displayed symbolically. Enemy units that can be seen by friendly forces are also shown as symbols. [Ref. 3: p. 5]

C. PEGASUS DATABASE

1. Background

The Pegasus database or perspective view database, covers a rectangular area of Fort Hunter Liggett which covers more than four hundred square kilometers. It is a graphical database created from aerial photographs, vegetation heights, and additional information required for perspective view generation. The database is organized into tiles, blocks and posts, with posts being the smallest element. The resolution considered for this thesis is a one square meter post. [Ref. 6: p. 3]

2. Thirty-two Bit Number Representation

Each post is described by a thirty-two bit binary number. There are nine elements to each 32 bit number, as shown in Figure 2 [Ref. 6: p. 3].

```
+---------------------+-+---+-------+-------+---+-+-+-------------+
| 3                   |2|   |       |      1|   | | |             |
|1 0 9 8 7 6 5 4 3 2 1|0|9 8|7 6 5 4|3 2 1 0|9 8|7|6|5 4 3 2 1 0  |
+---------------------+-+---+-------+-------+---+-+-+-------------+
|                     |E|   |       |       |   |N|S|             |
|         ELE         |L|UCI|  NOR  |  VGH  |VID|A|S|     GSV     |
|                     |2|   |       |       |   |T|B|             |
+---------------------+-+---+-------+-------+---+-+-+-------------+
```

Element Code	Number of bits	Maximum Value	Description
ELE	11	2047	Elevation in meters
EL2	12	4095	Elevation in half meters
UCI	2	3	Under Cover Index
NOR	4	15	Surface Normal Indicator
VGH	4	15	Vegetation Height Index
VID	2	3	Vegetation ID
NAT	1	1	Nature Bit
SSB	1	1	Sun Shade Bit
GSV	6	63	Gray Shade Value

Figure 2: Pegasus Database Elements

Visualization of each element is detailed in Figure 3. Grid elevation is defined by ELE and EL2. The NAT element defines whether or not the feature is manmade or natural. If feature is natural, value is 1. The UCI element is the height above ground of a feature. The NOR element gives an indication of the surface normal. The VGH element is the feature height. The VID element when combined with some of the other elements is used to determine exactly what the feature is. Colors can be represented by the GSV element, and passage of the sun defined by SSB element. [Ref. 6: p. 3] This 32 bit number is stored as a decimal and must be manipulated to extract the desired data.

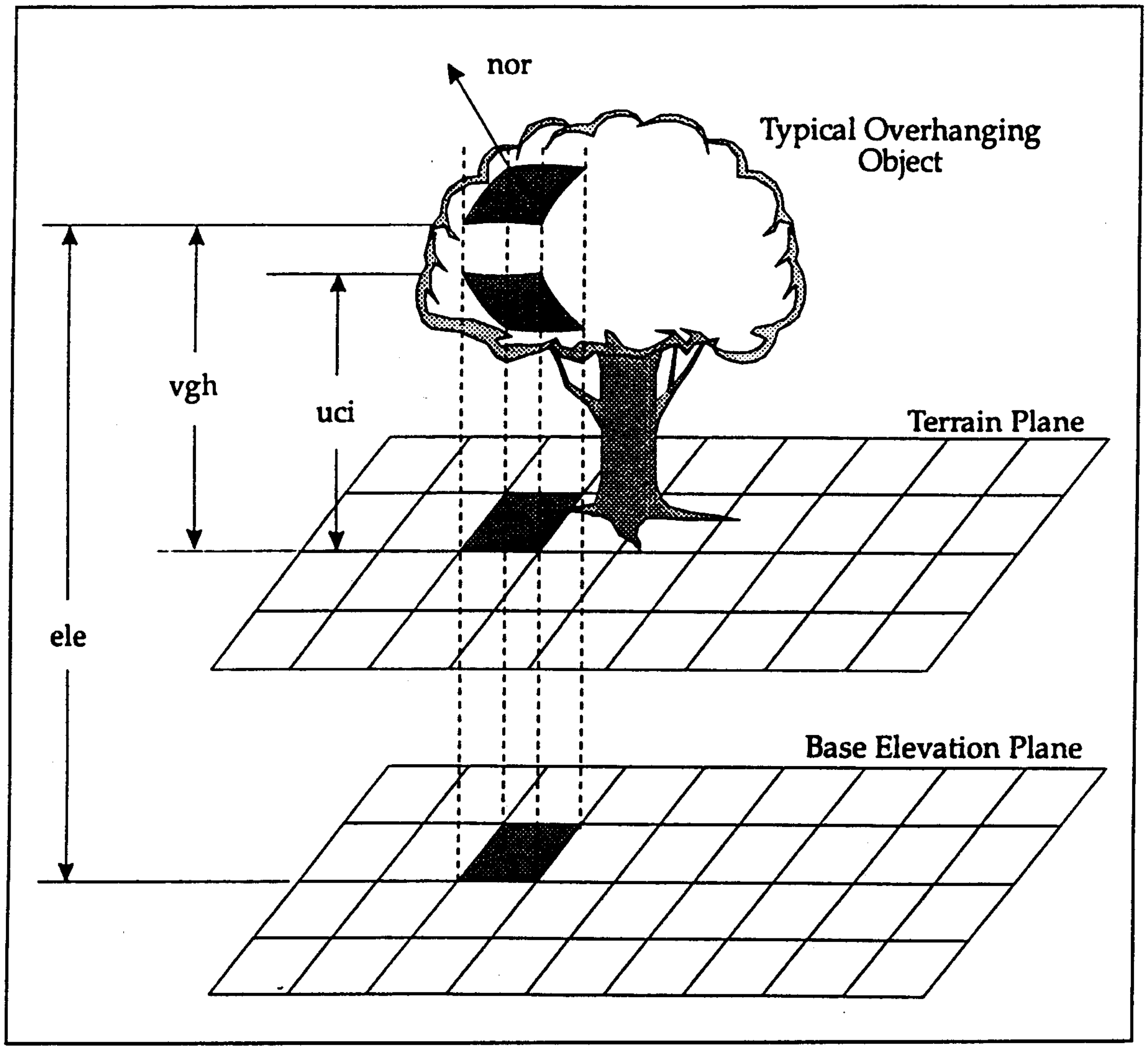


Figure 3: Visualization of Pegasus Database Elements

3. Grid Reference Position

The reference position of a grid can be determined by manipulating the array indexing. As an example, Figure 4 shows a sample 4x5 grid (20 grids, 20 numbers), and Table 1 shows a 1x20 array called locate[ ] which represents the 32 bit numbers describing each grid. To locate the decimal number describing grid position "A" in locate[ ], the following array location reference method is used:

$$\text{locate}[x * 5 + y] = 2099378$$

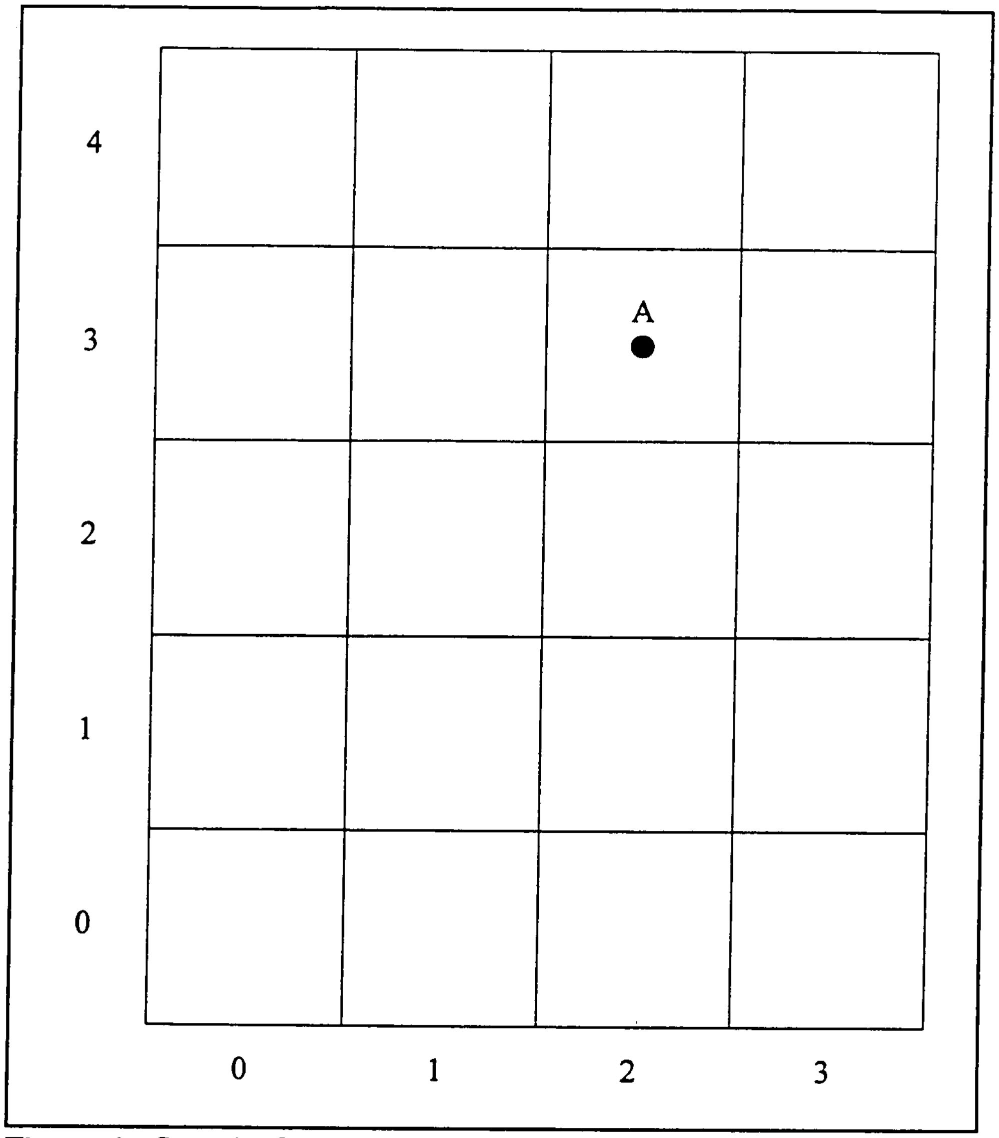


Figure 4: Sample Grid

This method starts counting (with zero) at the bottom left grid and counts up the first column, starting again at the bottom of the next column till grid position reached. In the example, position "A" is the thirteenth number of the array (2099378).

1202
1202
1202
1202
1202
1202
1202
1202
1202
1202
1202
1202
1202
2099378
1202
1202
1202
1202
1202
1202

Table 1: Locate[ ]

D. IMPROVED TARGET ACQUISITION ALGORITHM

1. Objective

The objective of the improved algorithm was to develop an improved method of target detection and acquisition which utilizes the Pegasus database. By being able to use Pegasus, the algorithm should be able to utilize the higher resolution and actually be able to determine if lines of sight exist and their quality. [Ref. 2: p. 16]

2. Accomplishments

a. Target Representation

With higher resolution of terrain representation comes the ability to have more than one LOS. The target used consists of a square three meters by three meters by one meter, and a one by one by one meter block sitting on the top center, as shown in Figure 5 [Ref. 2: p. 28]. Dependent on relative position of sensor and target up to eight faces of the target are available for lines of sight.

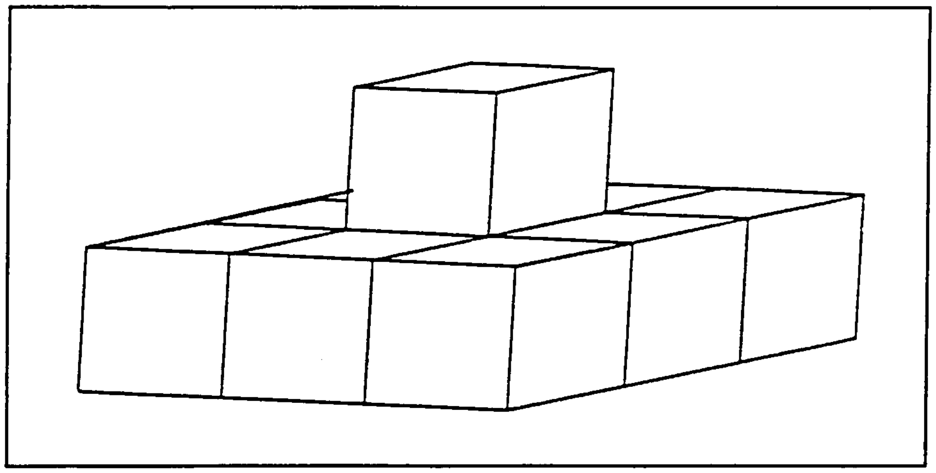


Figure 5: Target Model

b. Obstruction Representation

Obstructions are based on Pegasus style object representations. With the higher resolution, objects can have a wide variety of sizes and shapes. Individual trees, and buildings can be distinguished. Figure 6 [Ref. 2: p. 20] shows the representations used for obstructions.

c. Determination of Number of Possible Line's of Sight

From a sensor located at a fixed position on the terrain, and a target located at another position, subroutine **aspect()** is called. Based on the relative position of the target to sensor, the number of target faces that are presented to the sensor can be determined. There is a possibility of a maximum of eight faces or a minimum of four faces that can be presented at a time. By saying "visible" it is only meant that if the view between sensor and target is completely unobstructed, this is how many faces could be seen. The top surfaces to the target are not considered, nor are the back faces. [Ref. 2: p. 28]

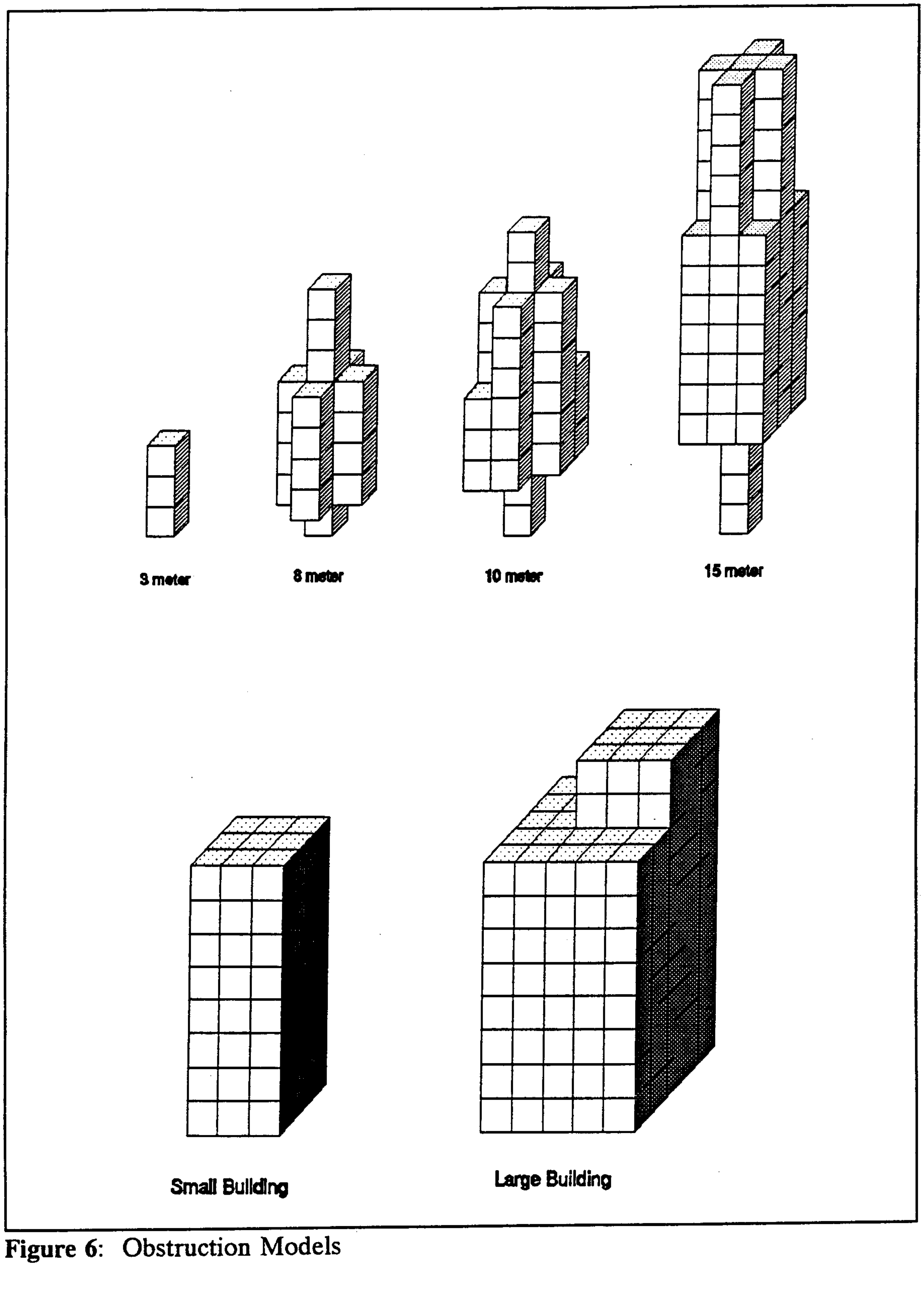


Figure 6: Obstruction Models

d. Stepping of Line of Sight

Depending on which distance is larger, the change in the x direction, or the change in y direction, one meter steps are taken in that direction. A single block is passed through on each step. Figure 7 shows an example of when the difference in x is larger than the y, therefore the LOS steps in the x direction one meter at a time. At each x coordinate, the y value is found using the y slope and origination point. As shown, the dark line represents the true path of the LOS, the shaded boxed show how the LOS steps. [Ref. 2: pp. 29-30]

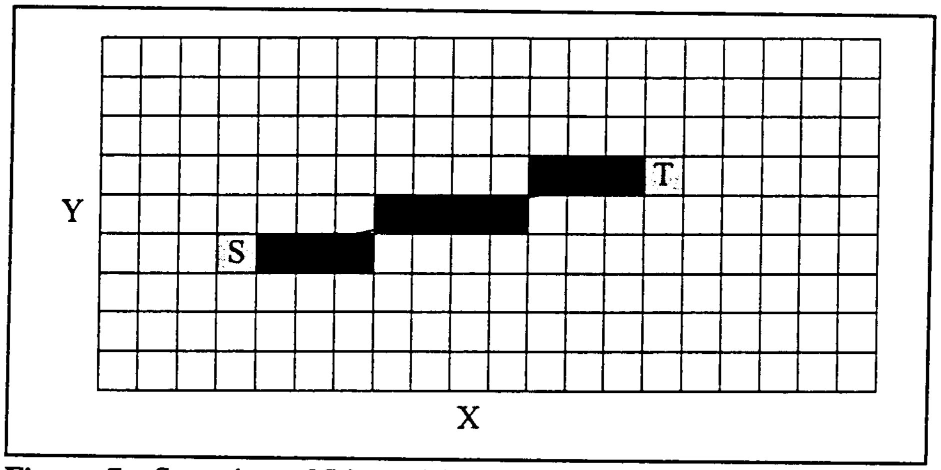


Figure 7: Stepping of Line of Sight

E. OBJECTIVE

The goal of this thesis is to add visualization to the improved target acquisition algorithm. By doing this, the data results of the algorithm can be visually confirmed. It will also show that it is possible to incorporate existing software into a training application with minimal programming experience. There are three major-steps that need to be taken in accomplishing this goal. They are: conversion of FORTRAN to C; understanding WorldToolKit (WTK); and creating a simulation incorporating the improved algorithm.

1. Conversion

The conversion process will involve breaking down FORTRAN code into specific functions and correlating these function to C functions. Each C function will then be tested as a separate program and then added to a main program.

2. WorldToolKit

WTK will require a thorough understanding of its many functions. Specifically for this project: how to create a terrain with objects on it; create objects; and be able to move them along the terrain. Additionally, an understanding of how to manipulate viewpoints, lighting, sensors, and windows will be required.

3. Simulation

This will involve combining WTK into the algorithm such that the visual representation of the terrain and motion of the target is driven by the improved algorithm. The simulation should have a target moving along an obstruction constrained terrain and display target data as the target moves.

II. CONVERSION OF CODE AND DATA

A. OBJECTIVE

The objective in converting the improved target acquisition algorithm from FORTRAN to C is to enable incorporation of WTK (programmed in C) into the improved algorithm. To simplify further discussions, the improved algorithm will hereafter be referred to as either FORTRAN code or C code.

The objective in converting 32 bit number data is to be able to manipulate specific elements. These elements are used to compare LOS with terrain features.

B. CODE CONVERSION

1. C Programming Basics

In order to begin conversion from FORTRAN to C, an understanding of a few basic C programming concepts is required. These basic concepts are as follows: preprocessor directives; functions; variables and constants; and pointers. The sample C program shown in Figure 8 contains examples of each of these concepts.

```
/* Example of Preprocessor Directive */
#include <stdio.h>

/* Example of Defining a Variable */
int test;

main()
{
/* Example of a Function */
printf('\nAnswer = %d\n', test);

/* Example of Using a Pointer */
test1= &test;
}
```

Figure 8: Basic C Program

a. Preprocessor Directives

After a program has been written, the next step is to compile it. During compilation, preprocessor directives tell the compiler to accomplish specific tasks. In the example given previously in Figure 8, **#include** directs the merger of a disk file (specified between < >) into the source code by the compiler [Ref. 7: p. 90]. This disk file is called a header file, by convention it is annotated by the extension **.h**. When called by the compiler, header files help check code for syntax and errors. A required C header file is **stdio.h**, which stands for standard input-output [Ref. 7: p. 92]. **Stdio.h** tells the compiler to be prepared for input and output functions in the program. If utilizing math functions such as **sqrt()** and **pow()** another required header file is **math.h**.

b. Functions

A function is defined as something used to solve a particular programming task. An example of a function is shown in Figure 9 [Ref. 8: p. 28].

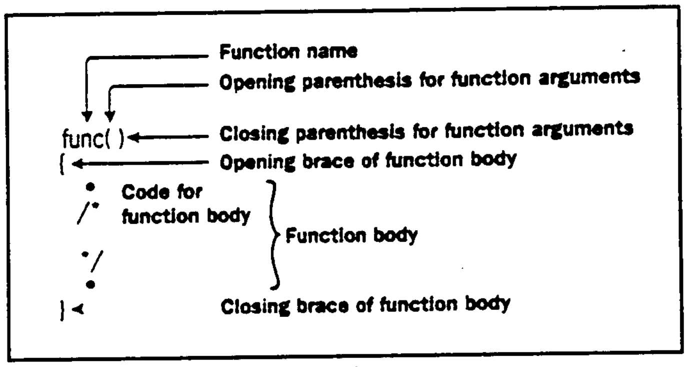

Function name
Opening parenthesis for function arguments
func( )
Closing parenthesis for function arguments
{
Opening brace of function body
/* Code for function body */
Function body
}
Closing brace of function body

Figure 9: Basic C Function

Every C program must have a **main()** function, which marks the point where program execution begins and ends. Within **main()**'s open and closed braces ({}), execution of program statements that appear first dictate what the program does first. Some of the standard functions utilized in the C code were **printf()**, **scanf()**, **ceil()**, **floor()**, **pow()**, and **sqrt()**. In addition to standard functions, specialized functions can be written to accomplish unique programming tasks. Some example of specialized functions used in the C code are: **terrain()**, **compare()**, and **aspect()**. Function arguments are enclosed between the parenthesis, if any are required. An argument is used to pass to the function anything it needs to complete its task. Examples of arguments are, variables, constants, and pointers. At the end of a function the statement **return;** is used to return control back to the calling function.

c. Variables and Constants

In order to correctly manipulate data within a program, it must be correctly defined within a program. Data is defined as either variables or constants. Variables hold different types of data such as characters, integers, or floating point decimals, and can be changed. Constants also hold different types of data, but remain the same throughout the program.

There are two types of variables, global and local. Global variables are defined outside a function. Problems can arise if a variable is defined as global and the same name is used in different program locations, but with different meaning. The value of one can be inadvertently changed, causing incorrect answers. Local variables are defined within the opening brace of the function, and will only be used in that function. [Ref. 7: pp. 298-299]

Local variables can also be defined as automatic or static. The default for local variables is automatic. Automatic means that when a function returns to the calling function, local variable values within it are erased. Static means that local variables will retain their value within a function when it returns to the calling function. [Ref. 7: p. 312]

d. Pointers

A key aspect to programming in C is the capability of variables pointing to addresses of other variables. These pointers are variables in themselves and follow the same rules which govern variables. So, instead of holding data values, which takes up memory, pointers provide a means of accessing and changing data by pointing to the address of another variable. There are two pointer operators in C, the **&** and *. The **&** operator is used to point to the memory address of whatever it precedes. The * operator either defines a pointer or dereferences the value of the pointer. Figure 10 illustrates how to use these operators. [Ref. 7: pp. 447-450]

```
/* Defines Pointer Called "ptr" */
int *ptr;

/* Defines "test" as an Integer with Value 10 */
int test=10;

/* Points "ptr" to Address of "test" */
ptr= &test;
```
Figure 10: Pointer Use Example

2. FORTRAN to C Conversion Procedure

a. FORTRAN Code Description

The FORTRAN code [Ref. 2: pp. 106-118] is organized into a main program with one subroutine called **aspect()**. A flowchart for the FORTRAN code is shown in Appendix A.

The following is a brief description of the logic steps within the FORTRAN code. The program starts by declaring variables, reading data from an external file into array tile( ), and manipulating this data into decimal form. There are ten target locations stored in array locate( ). DO loop (1) is entered with the maximum number of iterations being ten (ten target locations). After retrieving the first target location from locate( ), subroutine **aspect()** is called. Within **aspect()**, the number of target faces visible and location of these faces, is calculated. These are returned to the main program as values of "n" and array vistgt( ). Next DO loop (2) is entered with the maximum number of iterations being "n". Within this DO loop, an IF loop is entered. IF the slope of the line between target and sensor is greater in the x direction then **xstep** operations are performed. If slope is greater in the y direction, then **ystep** operations are performed. Depending on which was used in a particular iteration, both **xstep** and **ystep** operations enter another IF loop. IF a LOS exists then calculations are performed and DO loop (2) is re-entered. IF the LOS is terminated due to an obstruction DO loop (2) is re-entered without calculations. After "n" faces of a target is tested, DO loop (1) is re-entered for the next target location and the procedure is repeated.

b. Conversion

Within the main section of the FORTRAN code there are four different functional areas. The functional areas are: header operations; stepping in x direction; stepping in y direction; and calculations. Each of these four areas and the subroutine were converted to C separately, tested, and then one by one incorporated into a main C program.

Initially it was hoped that there could be a line for line translation from FORTRAN to C. This became unreasonable due to some procedural differences in code translation. First, bitwise manipulation differed due to not having a straight translation for the FORTRAN function **ibits()**. In C, there are two bitwise operators called AND and SHIFT. Second, mathematical functions differed due to not having a straight translation for the FORTRAN math function **nint()**, which round up or down to the nearest integer. In C, two functions had to be set up utilizing function **ceil()** (rounds up) and function **floor()** (rounds down). Third, relational operator structures differed due to C not using numbered loops which can use a **continue** statement to return to the beginning of the loop. C uses a **goto** statement to return to beginning of a loop.

It became clear that with C, there was an opportunity to organize the C code into specific function calls. The areas that the FORTRAN code were broken into for translation, now became C functions. Additionally, other areas within the FORTRAN code main program were also broken into C functions, such as **compare()** and **display()**.

Testing involved injecting test print statements into the FORTRAN code at key locations. Then entering specific variable values and running that section of C code. Finally, comparing C code results with output from FORTRAN code.

c. C Code Description

Once each of the separate sections of C code were tested satisfactorily, each was individually incorporated into a main program. The final C code was then tested and compared to FORTRAN code results. This conversion was completed satisfactorily. See Appendix B for flowchart of C code organization.

The following is a brief description of the logic steps for the C code. The program starts by declaring variables, reading data from an external file into array tile[ ], and manipulating this data into decimal form. FOR loop (1) is entered with the maximum number of iterations being ten (ten target locations). After retrieving the first target location from locate[ ][ ], function **aspect()** is called. Within **aspect()**, the number of target faces visible and location of these faces, is calculated. These are returned to the main program as values of "n" and array vistgt[ ][ ]. Next FOR loop (2) is entered with the maximum number of iterations being "n". Within this FOR loop, an IF loop is entered. IF the slope of the line between target and sensor is greater in the x direction then **xstep** operations are performed. Within the **xstep** operations, functions **ninty()** and **compare()** are called. If slope is greater in the y direction, then **ystep** operations are performed. Within the **ystep** operations functions **nintx()** and **compare()** are called. Depending on which was used in a particular iteration, both **xstep** and **ystep** operations enter another IF loop. IF a LOS exists then calculations are performed (**continue**:) and FOR loop (2) is re-entered. IF the LOS is terminated due to an obstruction function **display()** is called and FOR loop (2) is re-entered without calculations (**finish**:). After "n" faces of a target is tested, FOR loop (1) is re-entered for the next target location and the procedure is repeated.

C. DATA CONVERSION

1. 32 Bit Number Conversion

The 32 bit numbers are stored as decimals and must be manipulated in order to extract the desired data. In C, the **AND(&)** and **SHIFT(>>)** functions make it possible to emulate the FORTRAN function **ibits()**. The **AND** function is a bitwise operator and makes a bit by bit comparison of the number presented. The **SHIFT** function performs a left or right shift of bits within a binary number. An example would be to extract the NAT element (which is the 7th element) from a 32 bit number located in array test[ ]. First, test[ ] is ANDed with 128 (bintest[ ]=test[ ] & 128). The number 128 being base ten equivalent of binary number 10000000. The result (bintest[ ]) will be a binary representation of the first 7 bits of the 32 bit number. This number is then SHIFTed right

7 bits (nat[ ]=bintest[ ] >> 7) leaving the bit that represents the NAT element. [Ref. 7: pp. 196-203]

2. Terrain Modeling

In order to test the C code visually with WTK, the battlefield used to test the FORTRAN code will be re-created. The terrain for this battlefield consists of flat ground, small trees, large trees, and a small building. Each of these objects has a known height and undercover index since the higher resolution Pegasus database was used. The actual creation of the battlefield will be discussed in a later chapter.

3. Program CONVERT.C Description

To facilitate the creation of the test battlefield, program CONVERT.C [Ref. 2: pp. 101-102] was also converted from FORTRAN to C. The flowchart for CONVERT.C is shown in Appendix C.

The following is a brief description of program CONVERT.C. The program starts by declaring variable and arrays. The user is then prompted to input how many 32 bit numbers to be created, this input is read as **ans** from the keyboard. FOR loop (1) is entered with maximum iterations of **ans**. Next, FOR loop (2) array t[ ] is initialized to zero. Next, FOR loop (3) is entered with iteration counter **i** and maximum number of iterations of thirty-one. For each **i** from 1 to 9 an IF loop is entered. For purposes of brevity only the first IF loop will be described. IF i equals 1 then variable bit0 equals 0 and the user is prompted to enter the value of GSV. This input is read from the keyboard as **rem**. If the number entered is equal to zero, then the next IF loop is entered. For every IF loop, if the number entered is not zero then the number is converted into binary. The result is a 32 bit binary number describing a one meter square of terrain.

III. WORLDTOOLKIT

A. OBJECTIVE

The objective with regard to the utilization of WTK is to create a virtual world which will look like the test battlefield. Within the virtual world it will be necessary to represent targets and obstructions both realistically and in the form that the C code sees them.

B. WORLDTOOLKIT PROGRAMMING BASICS

1. Overview

WTK is a commercially available software program developed by SENSE8 Corporation, and is essentially a library of C functions. It provides an opportunity to create three dimensional (3-D) simulations and models with limited C programming skills. This is accomplished by adding specific WTK functions and their arguments to C source code. These WTK functions are broken down into classes. Assigned to each function class are subroutines identified by a handle. An example of the graphical object class handle would be **WTobject_new**. All functions require objects of a class to be pointed to as the first argument. As an example, **WTobject *cube** creates a new object and returns a pointer to cube. Pointers were discussed in the previous chapter. Somewhere later in a program, **cube=WTobject_new()** would be used to create a new object. In the next several sections, the major classes and objects utilized in WTK for this thesis will be discussed.

2. Universe Class

In its most basic sense, the universe created in WTK is a container.[Ref. 9: p. 2-1]. Inside the universe, all objects created are either stored (static) or manipulated (dynamic). Since only one universe can exist at a time, pointers are not required as a first argument when creating or loading a universe. The universe function handles are what is used to control simulation loop repetition activity. An important part of the flexibility in using WTK is the user input to the universe action function as illustrated in Figure 11 [Ref 9: p. 2-8].

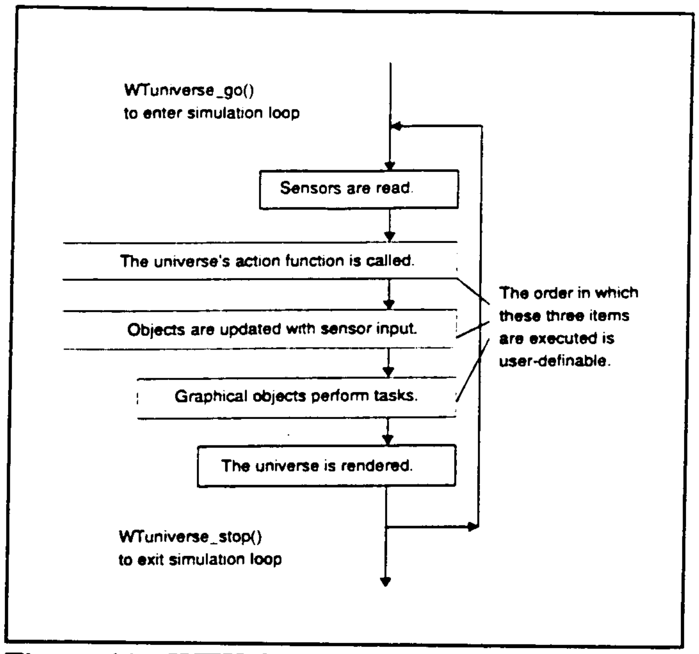

WTuniverse_go()
to enter simulation loop
Sensors are read.
The universe's action function is called.
Objects are updated with sensor input.
Graphical objects perform tasks.
The universe is rendered.
The order in which these three items are executed is user-definable.
WTuniverse_stop()
to exit simulation loop

Figure 11: WTK Simulation Loop

3. Graphical Object Class

Graphical objects are the building blocks of WTK [Ref. 9: p. 3-1]. They provide the user with many options in which to manipulate the simulation. Graphical objects can, interact with the user, interact with other objects, be hierarchial organized, have their motion effected by sensors, have tasks assigned, and have data associated to them [Ref. 9: p. 3-1]. For this thesis, objects were created from internal predefined object types using WTK functions, and by importing DXF format files generated from Autodesk. By convention, WTK uses the right hand rule for defining the world coordinate reference frame [Ref. 9: p. 3-7]. The world coordinate reference frame is fixed in space and therefore independent of any objects within the universe. However, a local coordinate reference frame can be created by determining the objects vertices and setting it to these planes. The following examples of graphical object class handles used in this thesis, **WTobject_getposition()**, **WTobject_newblock()**, **WTobject_newsphere()**, **WTobject_changecolor()**, and **WTobject_setposition()**.

4. Terrain Objects

Terrain objects represent landscape, and WTK can create three types, flat, random, and data generated. The test battlefield will be created as a flat 35x50 (square meters) terrain object. By default, terrain objects are given a checkerboard appearance to aid perspective view visualization. The default local reference frame for terrain objects is taken to be the same as world coordinate reference frame. Terrain object handle **WTterrain_flat()** was used to create the flat terrain test battlefield. [Ref. 9: p. 6-1]

5. Textures

Textures are equivalent to applying paint to surfaces of an object. This is done in order to improve realism and provide a means to conserve modelling labor and rendering speed by allowing a single object to be painted instead of modeling all of its 3-D details. Textures are obtained from video images or synthetically and are identified by the extension **.rgb**. Wood texturing was applied to cylinders to appear as tree trunks. Grass texturing was applied to spheres to appear as leaves. Building window texturing was applied to the small building to improve its realism. The texture handle **WTobject_settexture()** was used to apply textures to objects. [Ref. 9: pp. 8-1,8-2]

6. Light Objects

WTK has two kinds of lights, direct and ambient [Ref. 9: p. 9-1]. Ambient light is background light and it lights objects equally regardless of position or orientation. Directed light can be directed in a particular direction to produce shadows and contrast between objects. More than one light may be added to a simulation but, the more lights the greater impact on performance due to shading computations. The light object handles **WTlight_new()** and **WTlight_setambient()** were used to create a directed light source and set the ambient light source, respectively.

7. Sensor Class

Sensors allow the user to interact with the simulation directly. They are used to generate positional and orientational data by reading inputs from the real world [Ref. 9: p. 10-1]. A Spaceball by Spaceball Technology was created as a sensor and used to move a viewpoint through the test battlefield. The sensor handle

WTsensor_setsensitivity() set Spaceball sensor sensitivity to a default value and **spaceball=WTspaceball_new()** defined the Spaceball as a new sensor.

8. Viewpoints

In a window, the universe is drawn from parameters set to a viewpoint [Ref. 9: p. 11-1]. The major parameters are defined as, position, orientation, and direction. They allow the user to place viewpoints anywhere in the universe and look in specific directions and orientations. The viewpoint handle **WTviewpoint_addsensor()** was used to attach a viewpoint to the Spaceball. To create a new viewpoint, the viewpoint handle **WTviewpoint_new()** was used.

9. Path Class

A path is a combination of nodes which store a specific locations position and orientation data [Ref. 9: p. 13-1]. An object or viewpoint can be attached to a path, and moved from node to node. The more nodes, the smoother the transition between nodes. There is the capability to interpolate between nodes, which is especially effective when the path follows a curve. Paths can also be recorded, edited, saved, and played back [Ref. 9: p. 13-1]. Path class handle **WTpath_load()** was used to load external path files identified by extension **.pth**, which were created by an external program. This external path program will be discussed in the next chapter.

10. Window Class

A window object is a region of the screen where viewpoints are to be displayed. Several windows can be created, each with a different viewpoint attached. These windows can be created and deleted at any time during the simulation. The location and size of the windows is specified in the function arguments. The window class handle **WTwindow_new()** was used to create new windows. [Ref. 9: p. 15-1]

IV. SIMULATIONS USING WORLDTOOLKIT

A. OBJECTIVE

The simulations objective is to demonstrate that WTK can be incorporated into the C code. In doing so, they will also show what it is C code does, and visually validate the output.

Creating simulations will comprise of two steps and two different test battlefields. First, converted programs CONVERT.C and BTLFLD_TERR.C will be used to create a test battlefield with ground, obstruction, and target data. Second, WTK will create a virtual world test battlefield which will be incorporated into C code.

Three simulations were created to meet required objectives. They are PROG1.C, PROG2.C, and PROG3.C. This chapter will describe methodology used to create each simulation and their major functions.

B. TEST BATTLEFIELD CREATION

1. Test Battlefields

The test battlefields used in this thesis are similar to the 35x50 grid on which ground, obstructions and target were placed in ONEMETER.F. The only difference being actual location of targets. Figure 12 shows grid (ground), location of obstructions, and locations of target used in C code. Since there are actually two types of test battlefields that need to be constructed, the following is a brief description of both.

a. C Code Test Battlefield

The C code test battlefield is a database of Pegasus numbers that C code will read, manipulate, and then use in comparison with calculations of LOS. This database is created in two steps using two different programs. From program CONVERT.C, 32 bit Pegasus numbers are created for flat ground, three meter tall trees with no undercover index, eight meter tall trees with undercover index of one meter, and an eight meter tall small building (trees and building were shown in Figure 6). These are the only obstructions considered for this thesis.

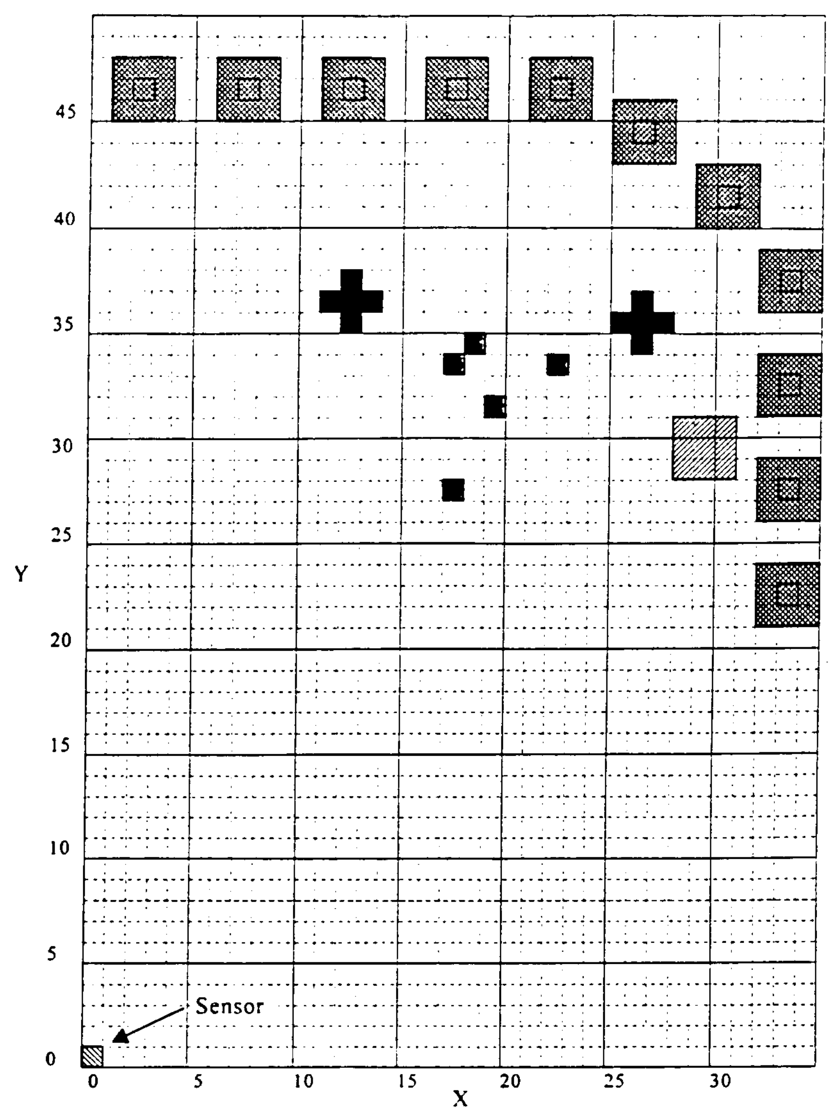


Figure 12: Test Battlefield

Program BTLFLD_TERR.C (also converted from FORTRAN to C) creates a 1x1750 (35x50) database array of Pegasus numbers obtained from CONVERT.C. Within BTLFLD_TERR.C, the entire database is first initialized as flat ground, then user is queried whether a three meter tree is to be placed on the flat ground. If answer is yes, then user is prompted to enter coordinate location. This is repeated until no more three meter trees are desired, same procedure is then repeated for eight meter trees and small buildings. For each obstruction entered, the flat ground 32 bit number is replaced by a new value. BTLFLD_TERR.C utilizes the array location referencing method previously discussed in Chapter 1 for placing appropriate 32 bit number in the database. Output from BTLFLD_TERR.C is written to an external data file called BTLFLD_TERR.DAT. The flowchart and code for BTLFLD_TERR.C are shown in Appendix F and Appendix G, respectively.

b. WTK Battlefield Terrain

The other test battlefield created is the one that WTK will visually display in the virtual world. From the library of WTK functions, the 35x50 flat grid (ground) is created. Located on the ground in same coordinate locations as C code battlefield, are same size trees and building. For different simulations, obstructions will be displayed differently. In PROG1.C and PROG2.C trees and building shapes will be created to appear as representative of real trees as possible. Additionally, in PROG1.C, textures will be applied to add realism. In PROG3.C, trees and building will be displayed as C code sees then, i.e., cubed entities.

2. Battlefield Targets

There are also going to be two types of target. C code will see the cubed target shown in Figure 5, and WTK will display a tank, shown in Figure 13 and the cubed target. To display targets, WTK loads a **.dxf** file to render the tank for PROG1.C and PROG2.C and a **.dxf** file to render the cubed target for PROG3.C.

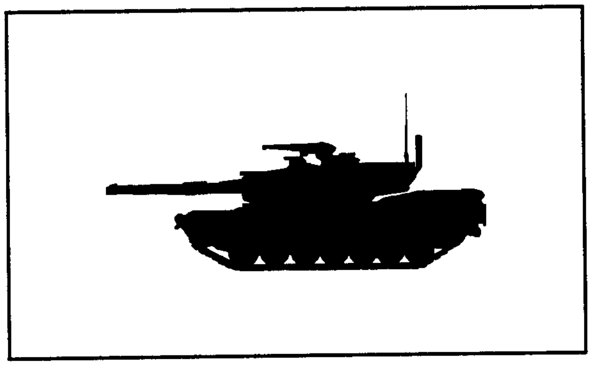


Figure 13: Tank Target

C. SIMULATION PROGRAMS DESCRIPTION

1. Overview

Each of the three programs created is essentially the same C code beginning, with WTK successfully incorporated as a function and LOS calculations imbedded within WTK. There are subtle changes to each program in their C code and each has major differences in what WTK displays.

Program PATH.C is an external WTK program that is used to create a user defined path. At screen prompts, user inputs path node locations and rotation. After entering all required nodes, user can select a method of interpolation to smooth out motion along any curves in path. Output from PATH.C [Ref. 10: pp. 69-75] is a file designated by extension **.pth**. The flowchart for PATH.C is shown in Appendix E. This program is used to create paths used in PROG2.C and PROG3.C.

Program PROG1.C utilizes an internal array of target locations (locate[ ]) which WTK uses to place the tank. The user manually presses a Spaceball button to move the tank to predetermined positions behind the trees and building, while displaying target data output at each prompt. There is an option to add viewpoints of universe from two additional windows, which can also be deleted at any time. The default window view is attached to the Spaceball, view from window one is from perspective of observer located at position (0,0), and view from window two is from back right corner looking toward sensor.

Program PROG2.C does not utilize an internal array of target locations, instead it uses WTK function **WTobject_getposition()** to obtain tank locations as the tank follows a predetermined path which was created using program PATH.C. The movement of the tank requires no user input other than to start motion by pressing a Spaceball button. In addition, output is continuously displayed to the screen. The same window options exist as with PROG1.C.

Program PROG3.C is similar to PROG2.C except it has a cubed target following a slightly different predetermined path created using PATH.C. Additionally, the default window viewpoint is different, and there is only option for one additional window. The path is essentially the same, but the cubed target does not have any rotation i.e., front face remains aligned with the x axis. The default view is from observer perspective located at position (0,0) and view from additional window is an overhead view.

The following three sections are a description of major functions that take place in each program. The flowchart and code for PROG1.C are shown in Appendix F and G, respectively. The flowchart and code for PROG2.C are shown in Appendix H and I, respectively. The flowchart and code for PROG3.C are shown in Appendix J and K, respectively.

2. Program PROG1.C

Program PROG1.C starts by declaring global variables, pointers, windows, and viewpoints. Function **main()** is entered and Pegasus database data is read into array tile[ ] from external file BTLFLD_TERR.DAT. This data is then manipulated by AND and SHIFT operators (previously discussed) ending with array tile[ ] being converted into various element arrays uci[ ], vghindex[ ], vgh[ ], vid[ ], nat[ ], and elev[ ]. These element arrays will be used to compare terrain data with each LOS to determine if LOS passes through a post. Function **terrain()** is then called, which for all practical purposes is WTK. The first action in **terrain()** is to call function **my_action()**. Within **my_action()**, each possible action prescribed by pressing a Spaceball button is defined. Simulation will begin if Spaceball button five is pressed, since this is the most important button, its actions will be detailed closer. Once button five is pressed, function **tgtxtyt()** is called to determine target position location by reading array locate[ ][ ]. Based on this data, a rotational value is assigned for rotating of target at a specific location. Next, function **los()** is called, with first action being to call function **aspect()** and determine number of faces presented for detection and location of these faces. After returning to **los()** with this data, each LOS from sensor to target face is stepped post by post. The direction of motion of LOS from post to post is determined in part by calculations involving functions **nintx()** and **ninty()**. In each target face LOS iteration, function **compare()** is called to compare LOS data to terrain data from various element arrays. The end result from **los()** is the number of LOS, range to target, percent of faces viewed, and area of target viewed. After returning to **my_action()** from **los()**, function **datatgt()** is called to display the LOS data to screen. Output data from PROG1.C is shown in Table 2. Control is then returned back to **terrain()**. Back in **terrain()**, function **tree()** is called and trees are created from data read from external data file TREE.DAT. Each tree consists of a stretched sphere attached to a cylinder. Texturing is applied to trees to enhance authenticity. After returning to **terrain()**, building is created, textured, and positioned, sensor is created and positioned, and tank is loaded from external file TANK.DXF. Finally, WTK creates universe and all objects in it. The next time Spaceball button five is pressed, universe data is updated to show tank in its new position and target data printed to screen is also updated. Simulation ends when Spaceball button eight is pressed at which time control is returned to **main()** and program PROG1.C ends.

		PROG1.C			PROG2.C			PROG3.C		
xt	yt	Range	%	Area	Range	%	Area	Range	%	Area
2	46	46.04	100	4.02	46.04	100	4.02	46.04	100	4.02
7	46	46.53	100	4.40	46.53	100	4.40	46.53	100	4.40
12	46	47.54	100	4.73	47.54	100	4.73	47.54	100	4.73
17	46	49.04	75	4.08	49.04	75	4.08	49.04	75	4.08
22	46	50.99	75	3.41	50.99	75	3.41	50.99	75	3.41
26	44	51.11	50	2.32	51.11	50	2.32	51.11	50	2.32
30	41	50.80	62	3.03	50.80	62	3.03	5.80	62	3.03
33	37	49.58	75	4.06	49.58	75	4.06	49.58	75	4.06
33	32	45.97	25	1.33	45.97	25	1.33	45.97	25	1.33
33	27	42.64	100	5.46	42.64	100	5.46	42.64	100	5.46
33	22	39.66	100	5.36	39.66	100	5.36	39.66	100	5.36

Table 2: Output Data

3. Program PROG2.C

Program PROG2.C starts by declaring global variables, pointers, windows, and viewpoints. Function **main()** is entered and Pegasus database data is read into array tile[ ] from external file BTLFLD_TERR.DAT. This data is then manipulated by AND and SHIFT operators (previously discussed) ending with array tile[ ] being converted into various element arrays uci[ ], vghindex[ ], vgh[ ], vid[ ], nat[ ], and elev[ ]. These element arrays will be used to compare terrain data with each LOS to determine if LOS passes through a post. Function **terrain()** is then called, which for all practical purposes is WTK. The first action in **terrain()** is to call function **my_action()**. Within **my_action()**, each possible action prescribed by pressing a Spaceball button is defined. Simulation will begin if Spaceball button five is pressed, since this is the most important button, its actions will be detailed closer. Once button five is pressed, the tank is attached to a path called **tnkpth**, this path is then started. Next, depending on data retrieved from **WTobject_getposition()** matching a predetermined location, function **tgtposn()** is called. Within **tgtposn()**, the targets position is determined, and function **los()** is called. Within **los()**, the first action is to call function **aspect()** to determine number of faces presented for detection and location of these faces. After returning to **los()** with this data, each LOS from sensor to target face is stepped post by post. The direction of motion of LOS from post to post is determined in part by calculations involving functions **nintx()** and **ninty()**. In each target face LOS iteration, function **compare()** is called to compare LOS data to terrain data from the various element arrays. The end result from **los()** is number of LOS, range to target, percent of faces viewed, and area of target viewed. After returning to **tgtposn()** from **los()**, function **datatgt()** is called to display LOS data to screen. Output data from PROG2.C is shown in Table 2. Control is then returned back to **terrain()**. Back in **terrain()**, function **tree()** is called and trees are created from data read from external data file TREE.DAT. Each tree consists of a stretched sphere attached to a cylinder. Texturing is not applied to the trees in this simulation. After returning to **terrain()**, building is created and positioned, sensor is created and positioned, tank is loaded from external file TANK180.DXF, and external tank path TANK.PTH is loaded.

Finally, WTK creates universe and all objects in it. The universe is updated continuously while the tank follows the path and the target data printed to screen is also updated continuously. The simulation end when Spaceball button eight is pressed at which time control is returned to **main()** and program PROG2.C ends.

4. Program PROG3.C

Program PROG3.C starts by declaring global variables, pointers, windows, and viewpoints. Function **main()** is entered and Pegasus database data is read into array tile[ ] from external file BTLFLD_TERR.DAT. This data is then manipulated by AND and SHIFT operators (previously discussed) ending with array tile[ ] being converted into various element arrays uci[ ], vghindex[ ], vgh[ ], vid[ ], nat[ ], and elev[ ]. These element arrays will be used to compare terrain data with each LOS to determine if LOS passes through a post. Function **terrain()** is then called, which for all practical purposes is WTK. The first action in **terrain()** is to call function **my_action()**. Within **my_action()**, each possible action prescribed by pressing a Spaceball button is defined. Simulation will begin if Spaceball button four is pressed, since this is the most important button, its actions will be detailed closer. Once button four is pressed, cube target is attached to a path called **tnkpth**, this path is then started. Next, depending on data retrieved from **WTobject_getposition()** matching a predetermined location, function **tgtposn()** is called. Within **tgtposn()**, targets position is determined, and function **los()** is called. Within **los()**, the first action is to call function **aspect()** to determine the number of faces presented for detection and the location of these faces. After returning to **los()** with this data, each LOS from sensor to target face is stepped post by post. The direction of motion of LOS from post to post is determined in part by calculations involving functions **nintx()** and **ninty()**. In each target face LOS iteration, function **compare()** is called to compare LOS data to terrain data from the different arrays. The end result from **los()** is number of LOS, range to target, percent of faces viewed, and area of target viewed. After returning to **tgtposn()** from **los()**, function **datatgt()** is called to display the LOS data to screen. Output from PROG3.C is shown in Table 2. Control is then returned back to **terrain()**. Back in **terrain()**, building is created and positioned, sensor is created and positioned, cube target is loaded from external file CUBETGT.DXF, and external tank path CUBE.PTH is loaded. The trees are then created from internal WTK object functions. Each tree consists of cubes arranged to represent Figure 6. Finally, WTK creates universe and all objects in it. The universe is updated continuously while the cube target follows the path and the target data printed to screen is also updated continuously. The simulation end when Spaceball button eight is pressed at which time control is returned to **main()** and program PROG3.C ends.

D. TERRAIN VARIATIONS

1. Existing Ground Elevation Representation

A limitation to simulation realism is WTK representing all ground elevations to be flat. Within WTK there are two additional ways of representing terrain ground, randomizing ground elevation heights and reading a data file to generate ground elevation heights. Both methods still produce a terrain grid that has a checkerboard appearance. The latter of these two methods is an improvement and was researched to try and develop a method of improving the reality of ground elevation visualization.

2. U.S. Geological Survey Terrain

The ground visualization improvement method researched involved trying to obtain existing ground elevation data from the U.S. Geological Survey (USGS). The USGS has two types of digital cartographic/geographic data files of possible interest. They are Digital Line Graph (DLG) files and Digital Elevation Model (DEM) files. Of the two types, DEM files represent the greatest potential for improving WTK ground visualization. Figure 14 shows an example of what a DEM can display. [Ref. 11: p. 1]

A DEM consists of an array of elevations that are referenced in the UTM coordinate system. Of several different size DEM files available, the 7.5 minute quadrangle was chosen. A 7.5 minute DEM is a 7.5x7.5 minute block of data generated from contour maps or scanning photographs. The data is organized from south to north in profiles that are then ordered west to east. The spacing between profiles is 30 meters. Figure 15 shows how a 7.5 minute DEM is organized. [Ref. 11: pp. 2-4]

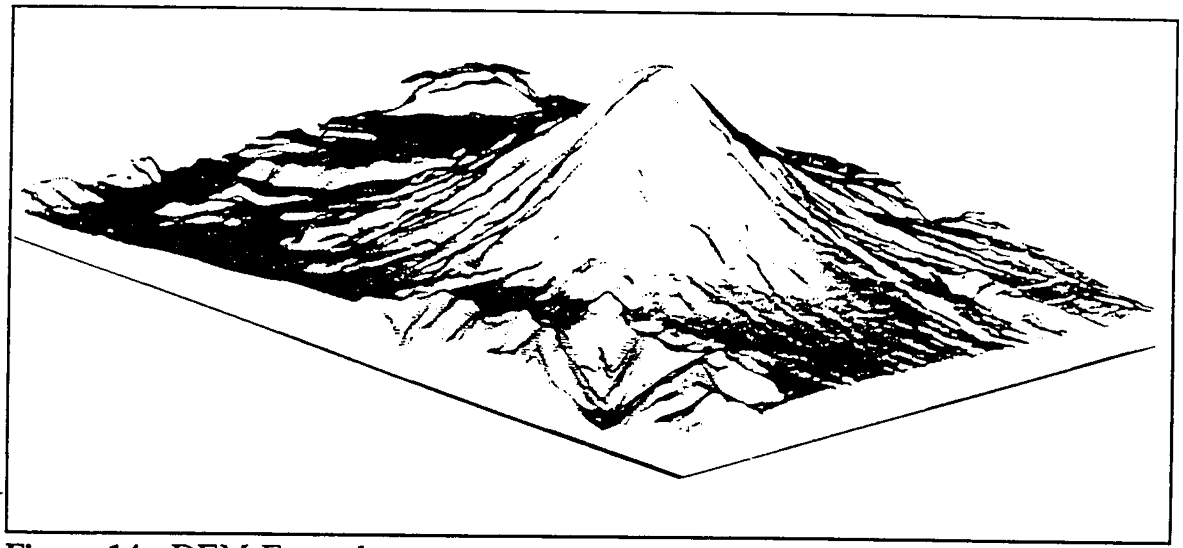


Figure 14: DEM Example

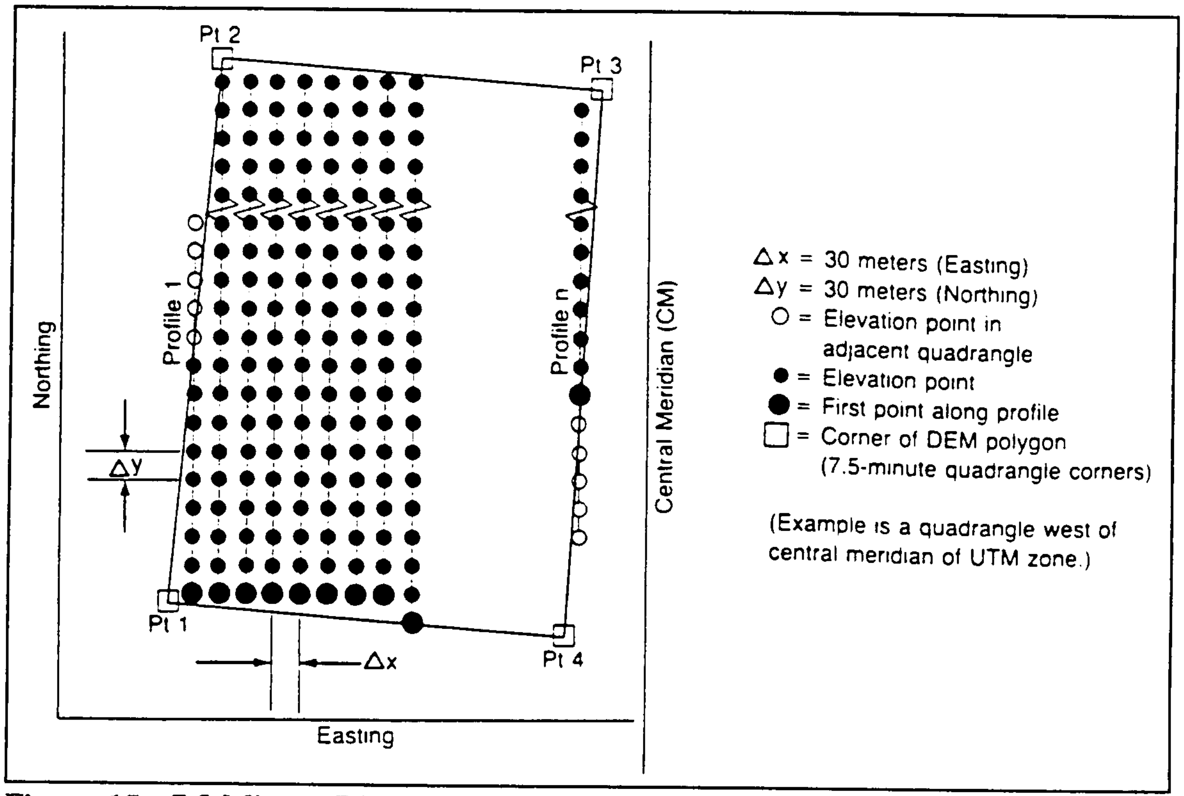

Pt 2
Pt 3
Pt 1
Pt 4
Profile 1
Profile n
Northing
Easting
Δy
Δx
Central Meridian (CM)
Δx = 30 meters (Easting)
Δy = 30 meters (Northing)
○ = Elevation point in adjacent quadrangle
● = Elevation point
● = First point along profile
□ = Corner of DEM polygon (7.5-minute quadrangle corners)
(Example is a quadrangle west of central meridian of UTM zone.)

Figure 15: 7.5 Minute DEM Organization

Each DEM is organized into three logical records, Type A, Type B, and Type C. Type A records contain header information about the DEM, including the number of

Type B records contained in the DEM. Type B records contain some header information about the profile followed by elevation data. Type C records contain statistical information about the accuracy of the data. The elevation information in the Type B records is what is needed by WTK. [Ref. 11: p. 15]

A 7.5 minute DEM was transferred from a USGS nine track magnetic tape onto an IRIS Indigo Workstation. A FORTRAN program was written to attempt to extract just the elevation data from the Type B records and put it in a format that WTK could read. Unfortunately due to time constraints, this effort was not able to be completed. The attempts to date had been unsuccessful.

V. DISCUSSION

This thesis has proven the capability of incorporating a commercially available visualization software program (WTK) into an existing training algorithm (C version of improved target acquisition algorithm). By successfully creating simulations with WTK incorporated into the C code, output data from the improved algorithm can now validated visually. The visualization of the improved algorithm also proves that the higher resolution Pegasus database can significantly improve the way Janus detects and acquires targets. This is accomplished by giving it the ability to have more than one LOS between sensor and target.

During the initial steps of learning C programming there were some lessons learned that are worthy of mentioning. First, verify that every C command ends with a semicolon (;). During compiling, errors will be created in lines below the missing semicolon. This gives an indication that there is a severe problem with a majority of the program, when in fact only one item is missing. Second, the importance of correctly defining variables as local or global cannot be understated. A thorough understanding of variables to be used and their interaction between different functions is a necessity. Finally, there must be a complete understanding of any unique functions that need to be translated. As an example, in order to translate the FORTRAN function **ibits()**, and understanding of binary numbers is required. This function first takes a decimal number and converts it to binary. Then depending on function arguments, extracts a specified number of bits, starting at a specified bit, and converts this binary number back into decimal form. The C equivalent to **ibits()** are the bitwise operators AND and SHIFT, previously discussed in Chapter II.

Programming in WTK required extensive use of the reference manual. The reference manual lists the library of functions within WTK and is very specific in their use. However, the proper arguments required to successfully utilize a function requires additional understanding of how WTK uses these arguments. As an example, if using **WTobject_move()** to move an object, one of the arguments is **WTpq** which is used to describe translation and rotation. If using **WTobject_translate()** to move an object, one of the arguments is **WTp3**, which describes an amount of relative motion. It can be confusing that the object is being moved in essentially the same fashion, however WTK needs the different arguments to complete the task.

An important factor in the realism of a simulation is object speed and smoothness of motion. In PROG2.C and PROG3.C both targets move along a path. Having the targets moving at a constant speed and without hesitation at nodes does add realism to the simulation. When moving along a path, the number of interpolation points between nodes directly impacts the smoothness of motion. Several factors effect the speed of objects moving along a path in WTK. The largest impact on speed comes from adding additional windows to the simulation. In PROG2.C and PROG3.C, if viewing target motion from just the default window, adding the sensor view window slows target motion significantly. To a smaller degree, applying textures to object faces slows speed down, as does loading a large object from an external file. There are ways to overcome the last two degradations in speed. When applying textures, only apply to sides of objects that will be seen. For loading large objects, in the arguments for loading an external object there is a scaling factor that can reduce the size of what is rendered. A factor in the speed not related to WTK directly is the speed of the workstation being used to run the simulation, the faster the better.

In the C code, only vertical faces of the target are considered for detection. Horizontal faces need to accounted for in both detection and as part of the total area presented for targeting. This is directly tied with the need to detect and acquire targets, moving on a non-flat terrain, in which horizontal faces may be presented for viewing, or with a sensor located above or below the target. Either way, an improved method of geometrically representing the target is needed to improve the C code to account for additional views of the target. A possible way to make this improvement is to change how targets are represented. At present, targets are not represented by Pegasus style numbers, but as fixed values within the C code. In order to more effectively use the C code, target representation could be added to the terrain description using Pegasus numbers.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

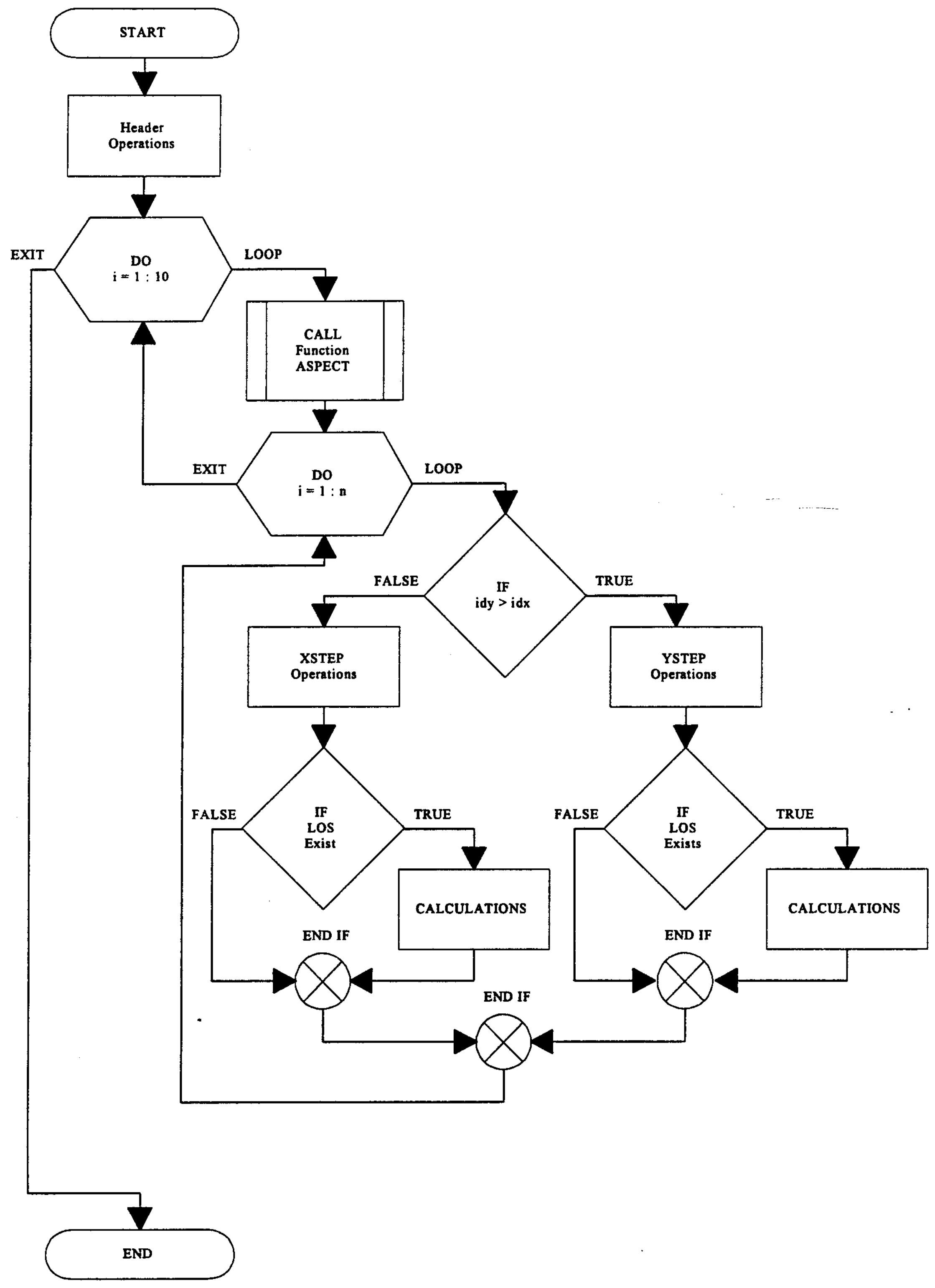
- Both C and WTK have the capability of being utilized with minimal prior programming experience.
- The improved target acquisition algorithm has been successfully converted into C.
- WTK has been successfully incorporated into the C version of the improved algorithm.
- Simulations created visually validate output data by showing there are more is one LOS when using a terrain described by the higher resolution Pegasus database.
- WTK can be used to add visualization to algorithms that produce positional information about an object.
- Visual validation of Pegasus capabilities to improve Janus by having more than one LOS. Improved algorithm proven to have potential to replace DOLOS subroutine.

B. RECOMMENDATIONS

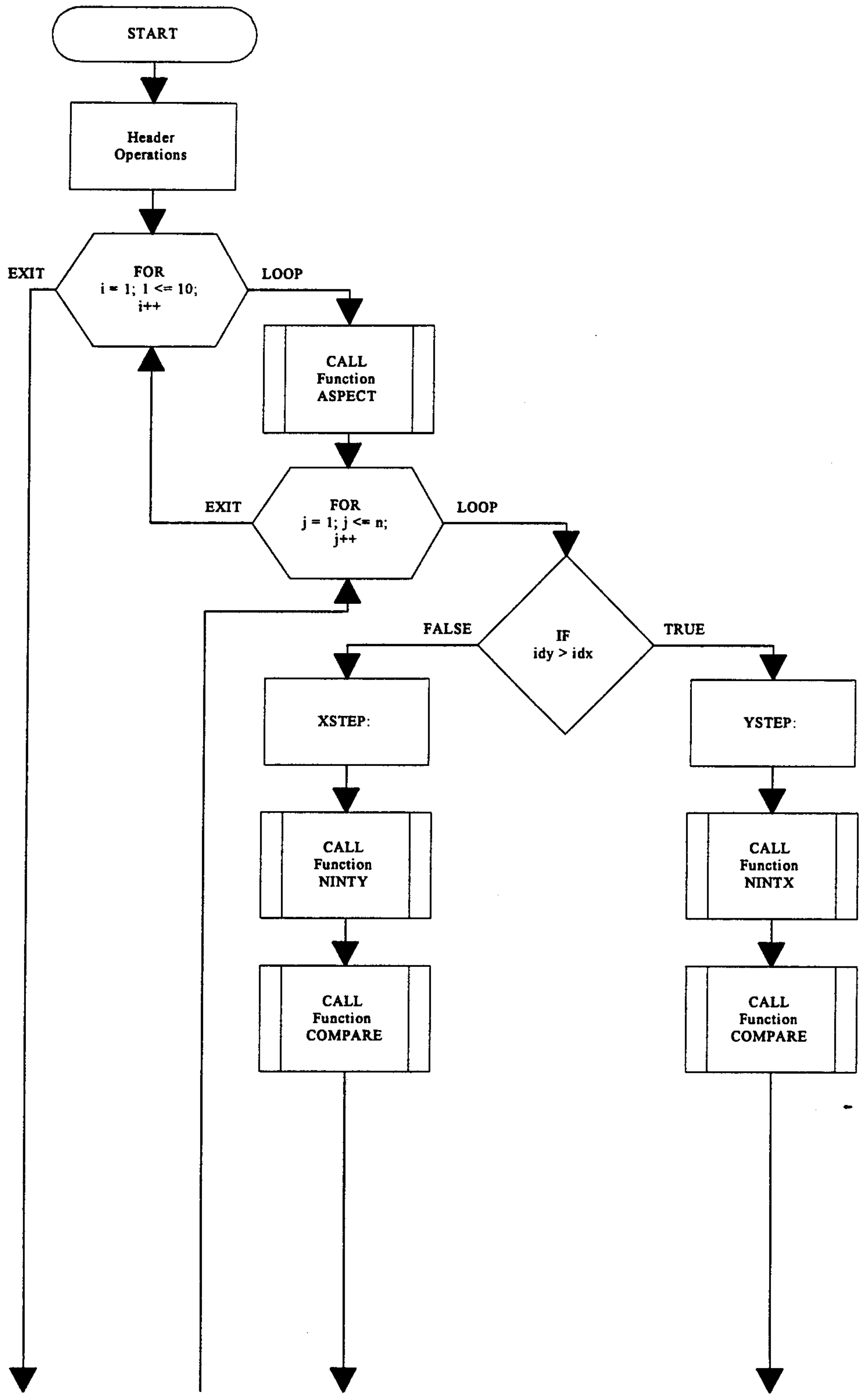
The use of WTK in the improved target acquisition algorithm has shown visually that the improved algorithm and Pegasus database could enhance Janus by improving scenario realism. The following are areas which could positively impact Janus:

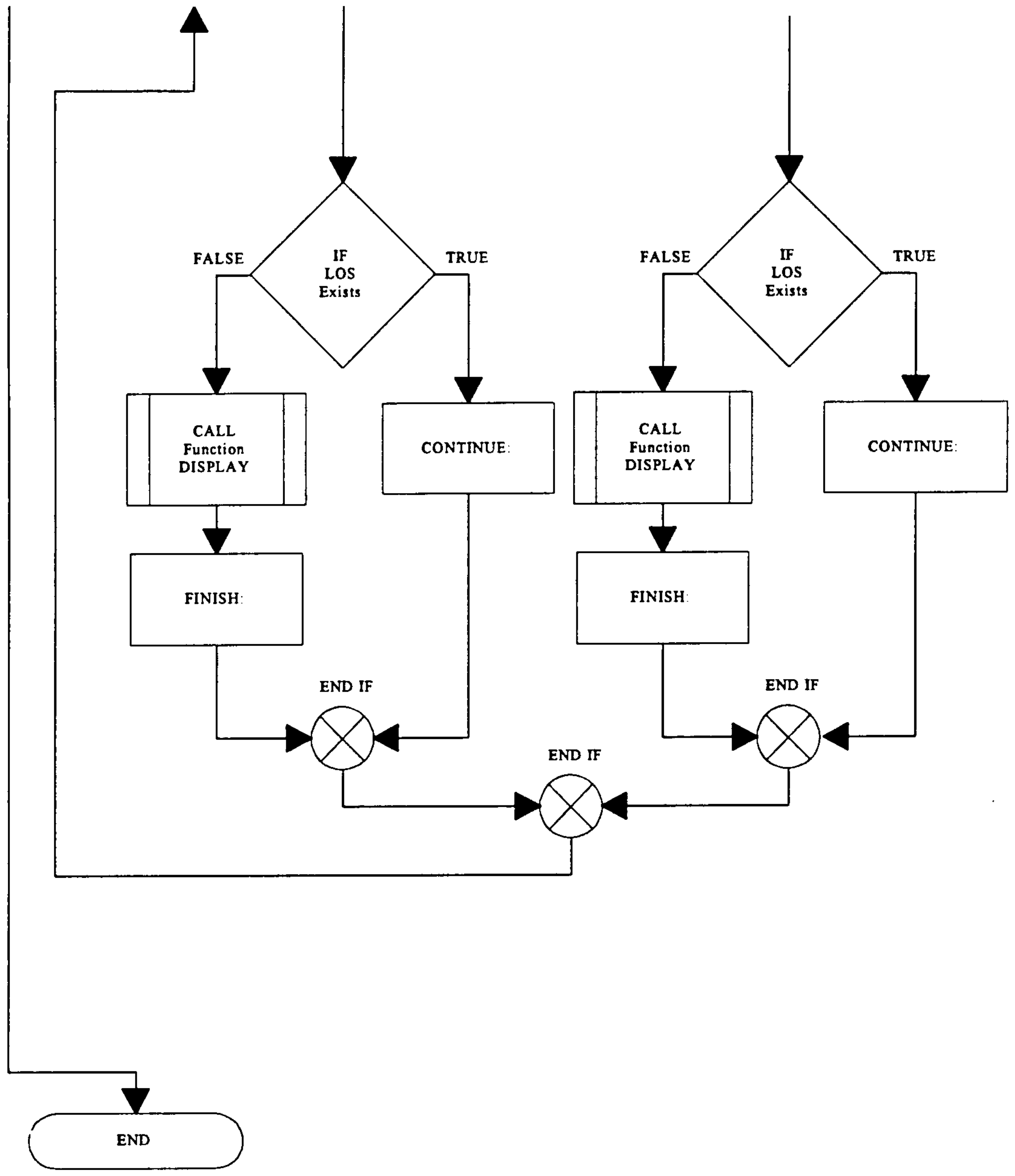
- Improve existing WTK simulations by, using non-flat terrain, targets represented by Pegasus numbers, and have both sensor and target moving.
- Incorporate improved algorithm into Perspective View Generator model.
- Incorporate improved algorithm into Janus.

APPENDIX A. FLOWCHART ONEMETER.F

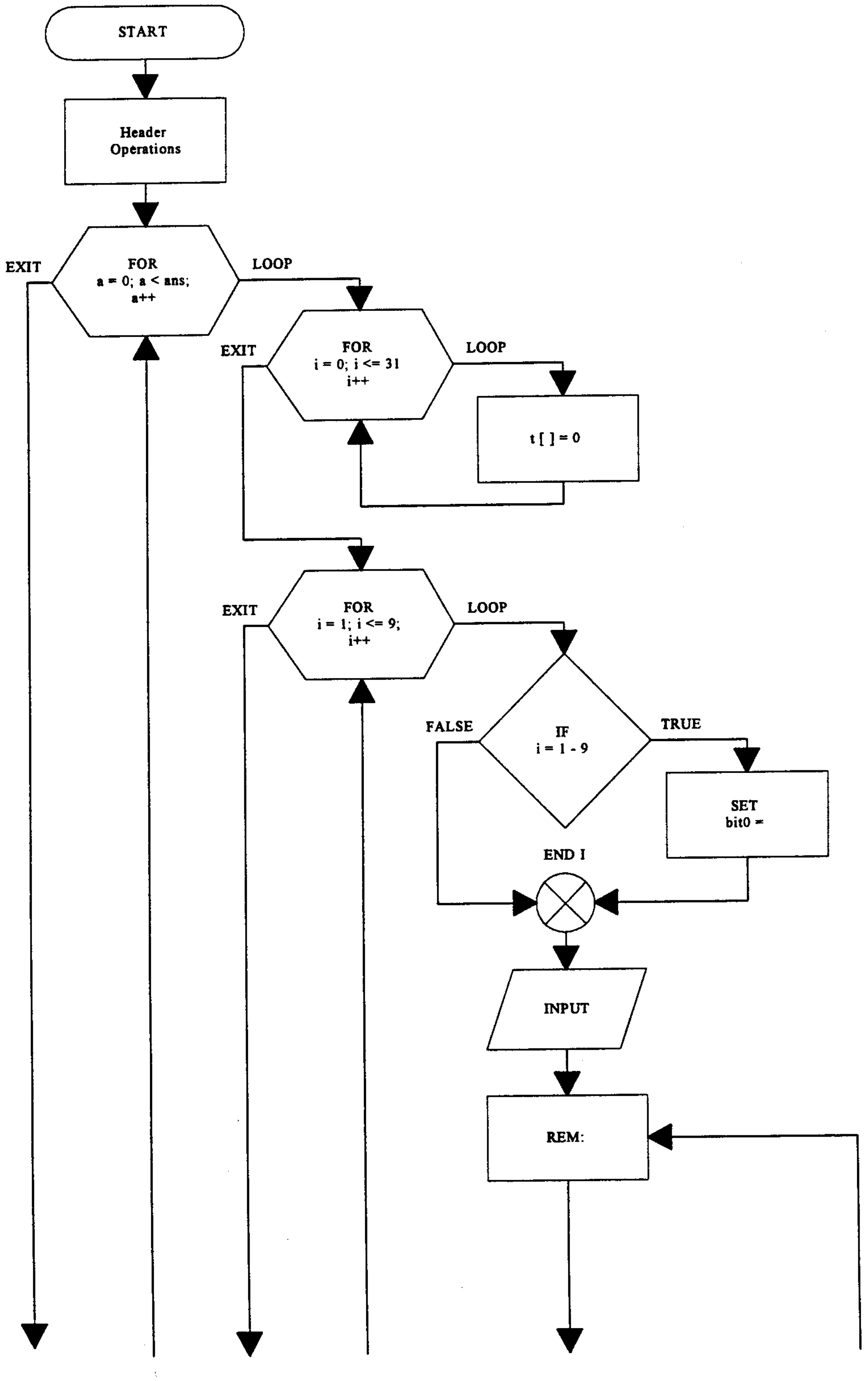


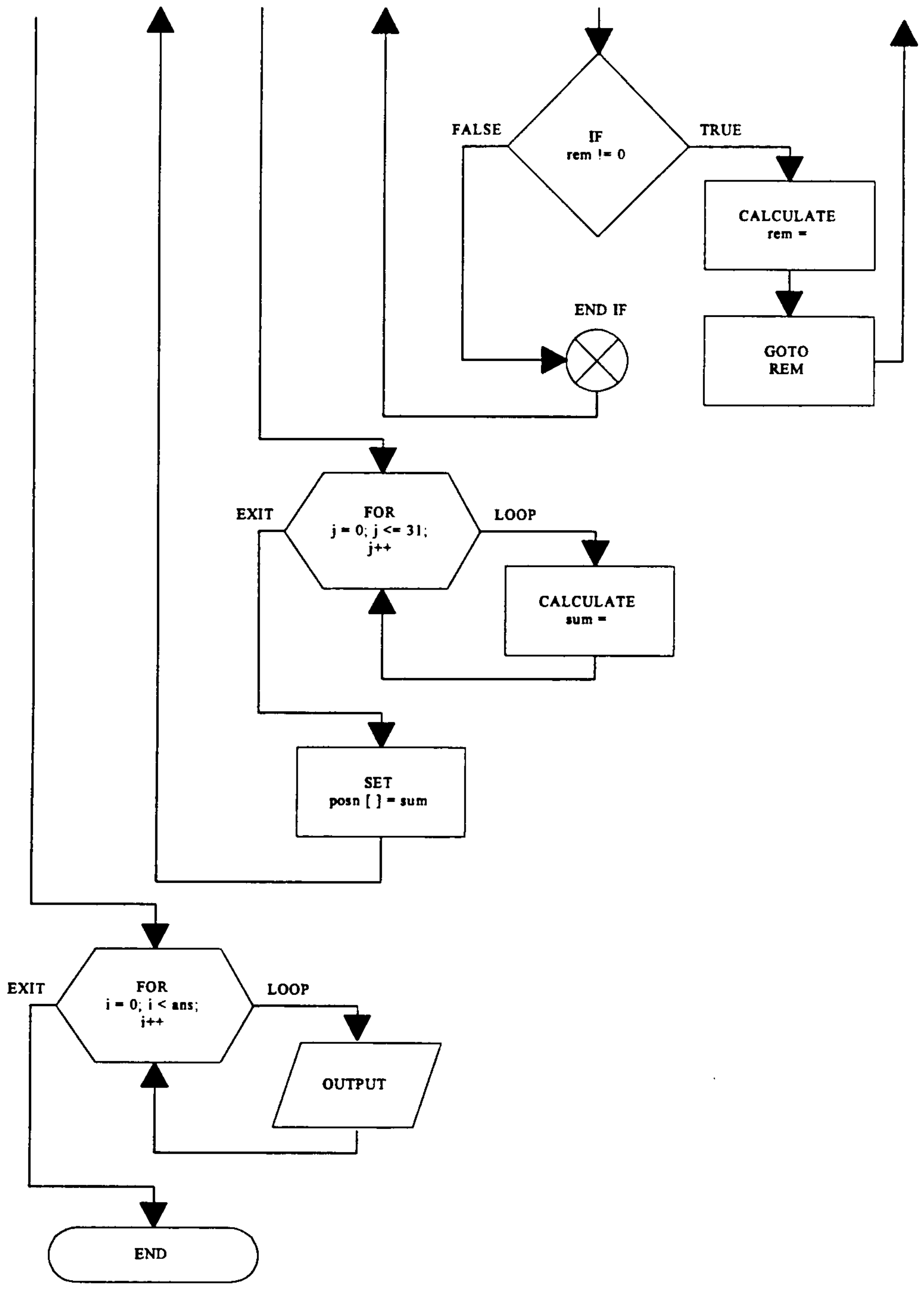
APPENDIX B. FLOWCHART C CODE




FALSE
IF LOS Exists
TRUE
FALSE
IF LOS Exists
TRUE
CALL Function DISPLAY
CONTINUE:
CALL Function DISPLAY
CONTINUE:
FINISH:
FINISH:
END IF
END IF
END IF
END

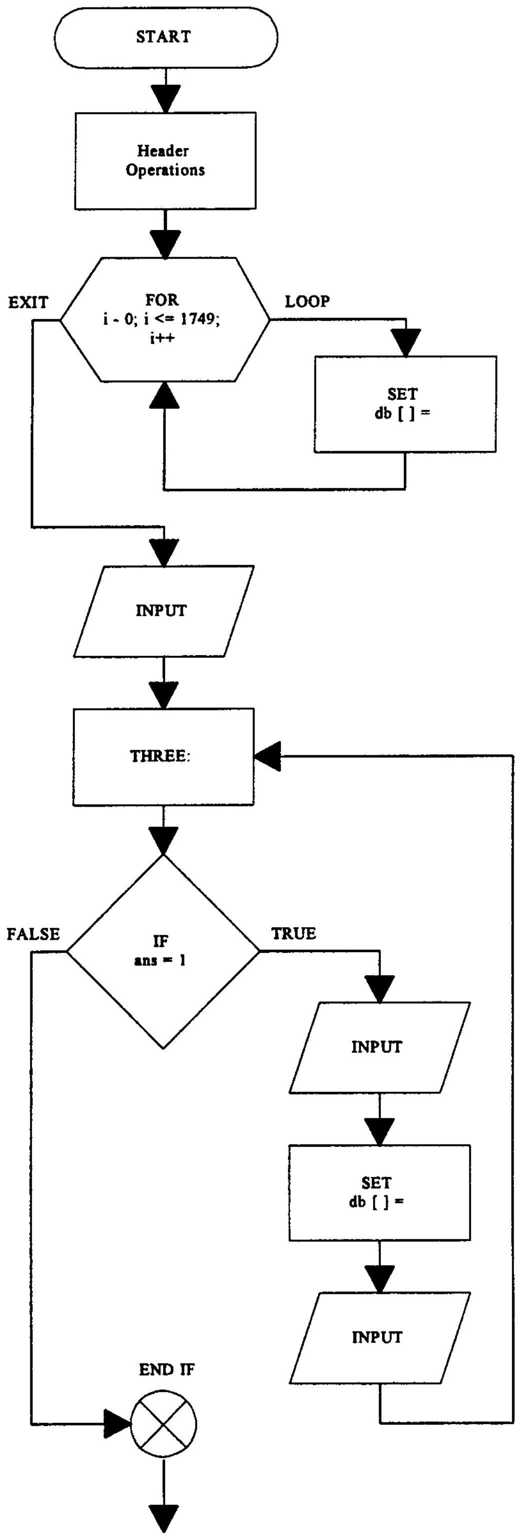
APPENDIX C. FLOWCHART CONVERT.C

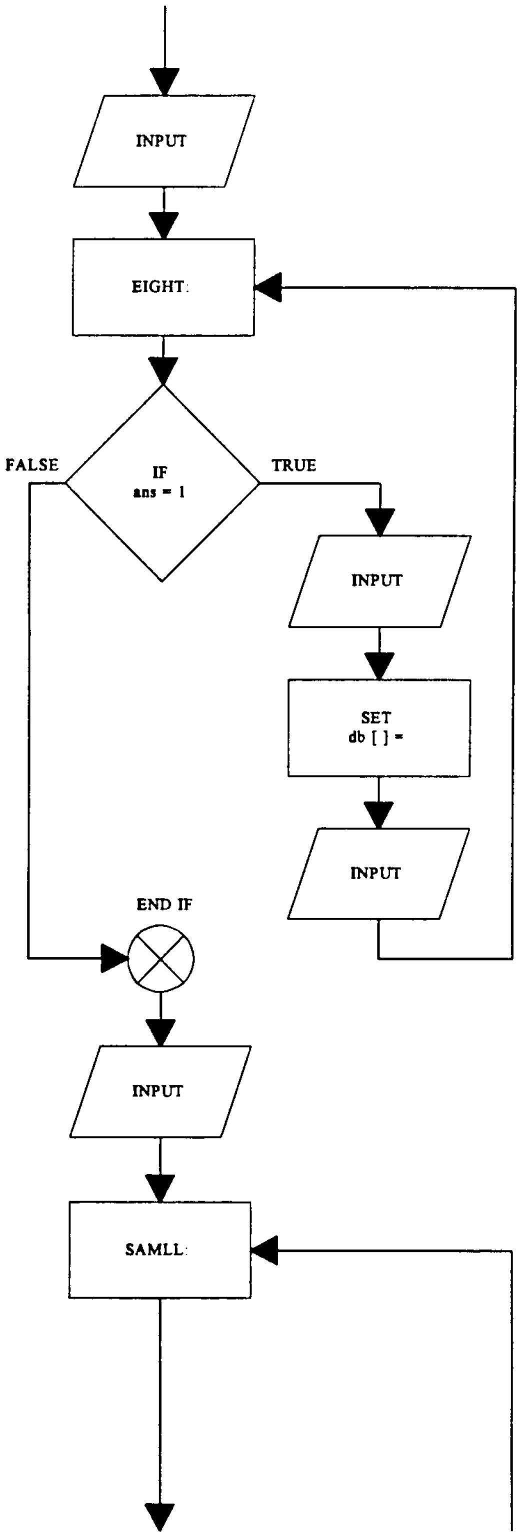


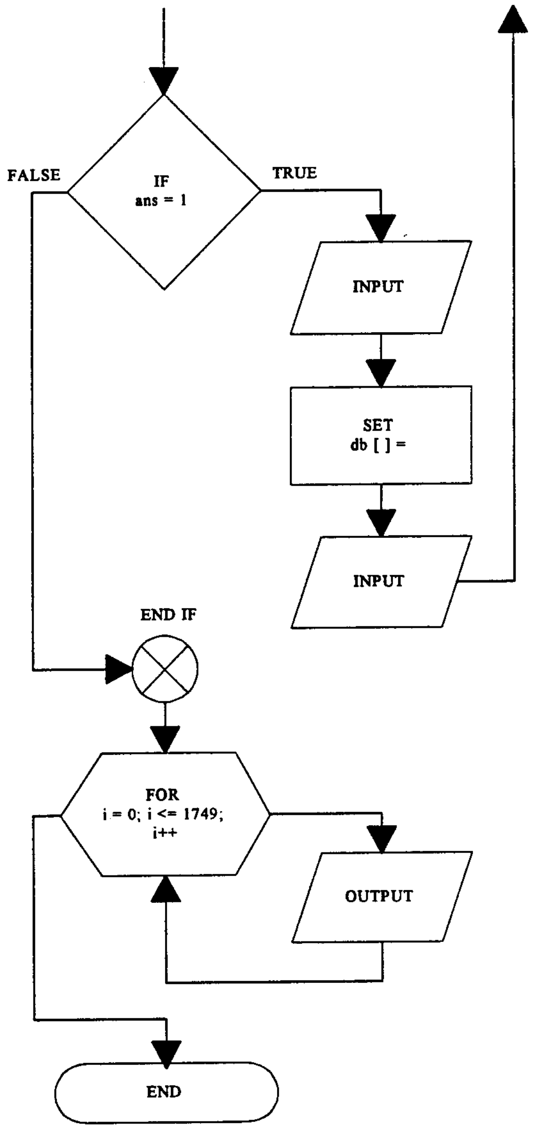


FALSE
IF
rem != 0
TRUE
CALCULATE
rem =
END IF
GOTO
REM
EXIT
FOR
j = 0; j <= 31;
j++
LOOP
CALCULATE
sum =
SET
posn [ ] = sum
EXIT
FOR
i = 0; i < ans;
i++
LOOP
OUTPUT
END

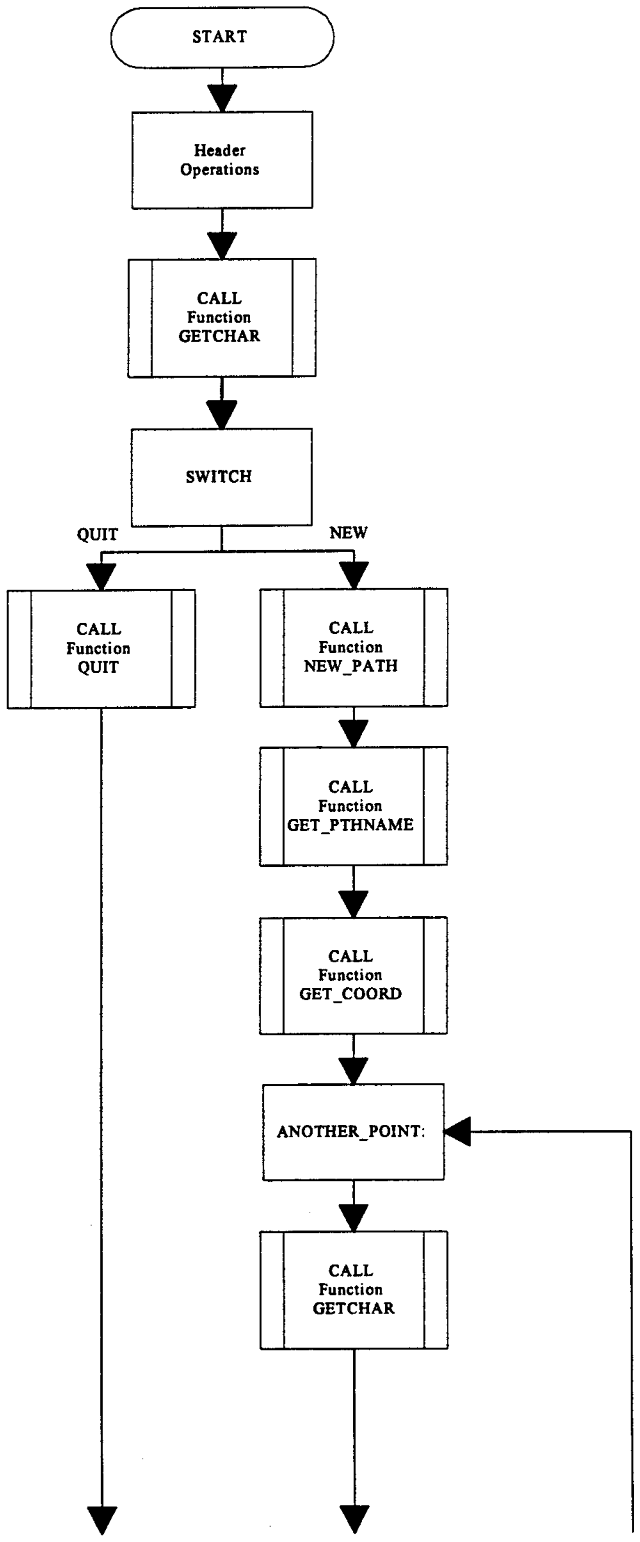
APPENDIX D. FLOWCHART BTLFLD_TERR.C

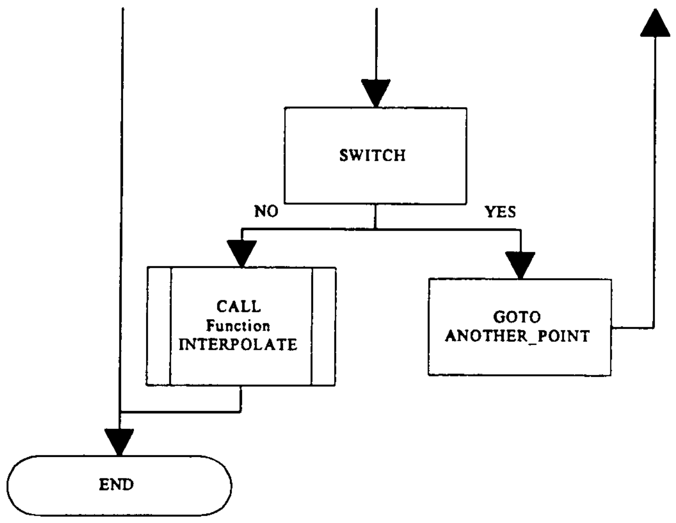



INPUT
EIGHT:
FALSE
IF
ans = 1
TRUE
INPUT
SET
db [ ] =
INPUT
END IF
INPUT
SAMLL:

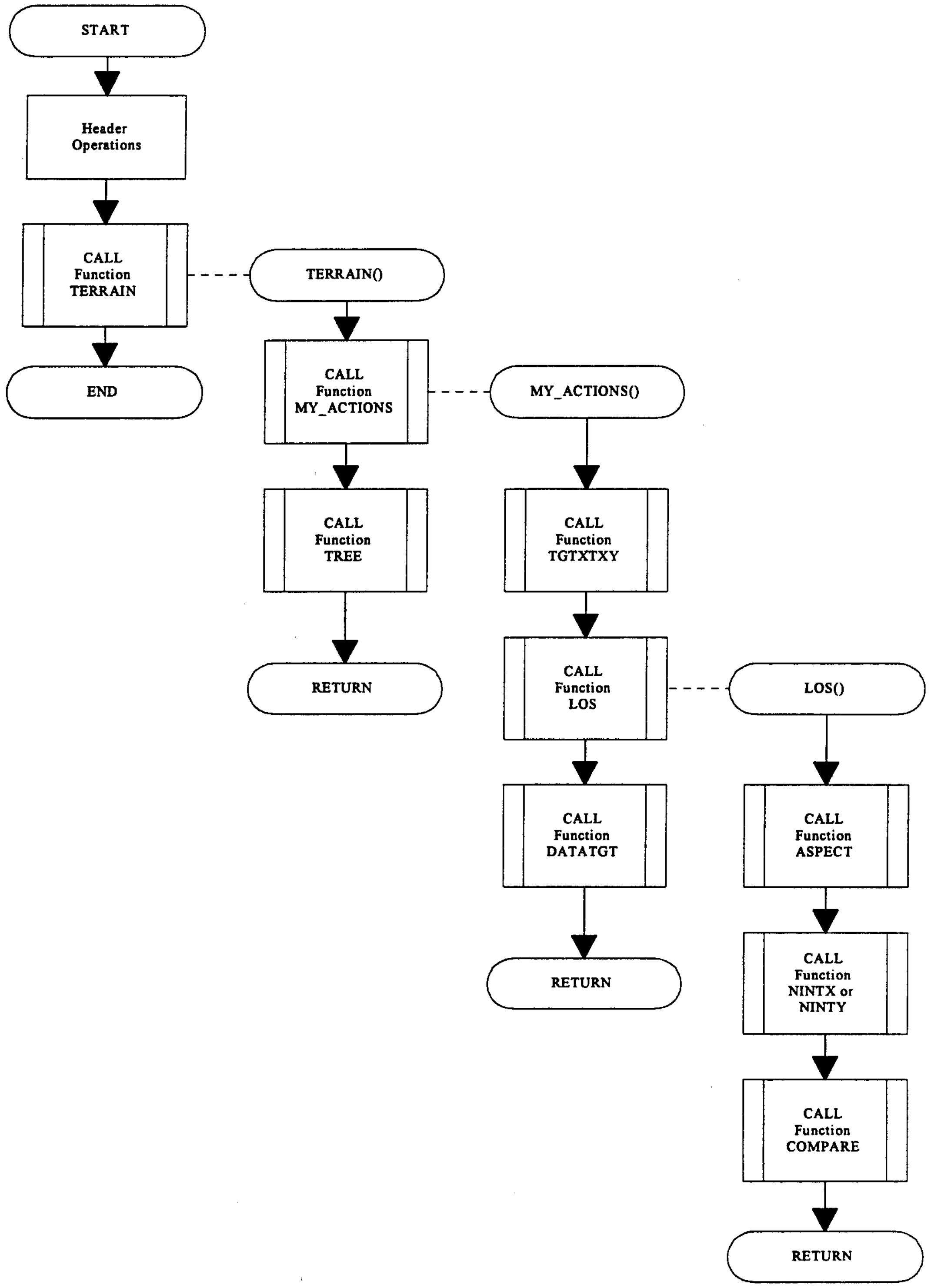

FALSE
IF
ans = 1
TRUE
INPUT
SET
db [ ] =
INPUT
END IF
FOR
i = 0; i <= 1749;
i++
OUTPUT
END

APPENDIX E. FLOWCHART PATH.C




SWITCH
NO
YES
CALL
Function
INTERPOLATE
GOTO
ANOTHER_POINT
END

APPENDIX F. FLOWCHART PROG1.C



APPENDIX G. PROGRAM PROG1.C

```
/* PROGRAM PROG1.C */
/*************************************************************************/
/* This program calculates Line Of Sight (LOS) data and visualizes a target moving*/
/* through a 35x50 grid using a database of Pegasus numbers and is designed to    */
/* incorporate attenuation by vegetation.  The main section of this program calls  */
/* function terrain() which is used to start WorldToolKit and create a universe    */
/* where there is a flat terrain with trees and a building located on it.  The trees and*/
/* building are represented as close approximations to real objects.  The simulation */
/* has the target moving behind the trees and building by manually pressing the     */
/* spaceball.  Each press of spaceball button FIVE moves the target one position.   */
/* Function terrain() calls function los() which is used to calculate target data.  */
/*************************************************************************/
/* Standard C and WorldToolKit header files */
#include <stdio.h>
#include <math.h>
#include"wt.h"
#include"wt.p"

/* Defines a pointer spaceball to structure type WTsensor */
static WTsensor *spaceball=NULL;
static WTsensor *mouse=NULL;

/* Calls function tree() but does not pass or return any values */
static void tree();

/* Defines windows and viewpoints */
static WTwindow *win1, *win2;
static WTviewpoint *objview1, *objview2;

/* Defines objects */
WTobject *bldg, *copy, *cyl, *original, *sensor, *sphere, *tank, *terrobj;

/* Defines pointers to WTp3, WTq, and WTpq structures */
WTp3 at,dp1,dp2,dp3,dp4,dp5,dir,factors,p,p1,p2,ppp1,ppp2,pt;
WTq pp1,pp2;
WTpq modelview;

/* Defines light object to be a pointer to type WTlight */
WTlight *mylight;
```

```
/* Initialize arrays and declares variables */
int ans,c,cxt,cyt,ho,ht,i,ix,iy,idx,idy,istart,istop,j,n,nolos,prcnt,xs,xt,xmax,xmin,xstart,ys;
int yt,y,max,ymin,ystart,zs,zt,zdirt,ztree,binelev[1750],binnat[1750],binuci[1750];
int binvghindex[1750],binvid[1750],elev[1750],locate[12][3],nat[1750],realvgh[10];
int tgt[17][5],tile[1750],uci[1750],vgh[1750],vid[1750],vghindex[1740],vistgt[9][9];
float aproj,atten,attenf,dx,dy,dz,denfol,dist,r,rad,rdx,rdy,row,rdns,range;
float totarea,visaproj,x,y,y_rotate,z,zx,zy,mdata[7][3];

/* Standard C file pointers */
FILE *fp;
FILE *fp1;

main()

{

/* Reads database information into array from data file "btlfld_terr.dat" */
        fp=fopen("btlfld_terr.dat","r");
        for (i=0; i<=1749; i++)

                {

                fscanf(fp," %d", &tile[i]);

                }

        fclose(fp);

/* Assigns 1 meter of foliage an attenuation of 30% of the LOS */
        denfol=0.3;

/* The vegetation height from Pegasus has values of 0-15, each of these values
   represents a particular height or range of heights. The following is used to correlate
   the vgh value to a meaningful height */
        realvgh[0]=0;
        realvgh[1]=0;
        realvgh[2]=1;
        realvgh[3]=2;
        realvgh[4]=3;
        realvgh[5]=4;
        realvgh[6]=5;
        realvgh[7]=8;
        realvgh[8]=10;
        realvgh[9]=15;
        realvgh[10]=20;
        realvgh[11]=25;
        realvgh[12]=30;
        realvgh[13]=35;
        realvgh[14]=40;
        realvgh[15]=47;

/* Once the database for the grid desired has been read into an array, the components
   of information are extracted from the 32 bit number using the AND and SHIFT
   functions. The groups of information of use in the program are placed in their own
   arrays for rapid recall during calculations as follows:
   elevation of highest point in grid        - ELEV
   under cover index                         - UCI
   converted vegetation height               - VGH
   vegetation ID                             - VID
   nature bit                                - NAT */
        for (i=0; i<=1749; i++)

                {

                binuci[i]=tile[i] & 786432;
                binvghindex[i]=tile[i] & 15360;
                binvid[i]=tile[i] & 768;
                binnat[i]=tile[i] & 128;
                binelev[i]=tile[i] & 4292870144;
                uci[i] = binuci[i] >> 18;
                vghindex[i] = binvghindex[i] >> 10;
                vgh[i]=realvgh[vghindex[i]];
                vid[i] = binvid[i] >> 8;
                nat[i] = binnat[i] >> 7;
                elev[i] = binelev[i] >> 21;

                }

/* This program is currently written to run in a stand alone mode.  Sensor information
   is entered from the keyboard, and target data is read from an array to simulate a
   moving target. Inputs sensor location and height */
        printf("\nThe Default Sensor Coordinates are xs = 0, zs = 0");
        printf("\nand hs = 1\n");
        printf("\nDo you wish to Change these values?  1=Yes, 2=No\n");
        scanf(" %d",&ans);

        if (ans == 1)
                {
```

```
            BEGIN:

            printf("\nEnter Sensor X Coordinate (X,Y)\n");
            scanf("%d,%d",&xs,&ys);
            printf("\nEnter Sensor Height\n");
            scanf("%d",&ho);
            printf("\nAre You Sure?  1=Yes  2=No \n");
            scanf(" %d", &ans);

        if (ans == 2)

            {

            goto BEGIN;

            }

            }
        else
            {

            xs=0;
            ys=0;
            ho=1;

            }

        printf("\nPress Spaceball Button 1 to add a window\n");
        printf("\nPress Spaceball Button 2 to add another window\n");
        printf("\nPress Spaceball Button 3 to delete second window added\n");
        printf("\nPress Spaceball Button 4 to delete first window added\n");
        printf("\nPress Spaceball Button 5 to move tank\n");
        printf("\nPress Spaceball Button 8 to end simulation\n");

/* Array containing target location, this is used to simulate a moving target */
        locate[1][1]=2;
        locate[1][2]=46;
        locate[2][1]=7;
        locate[2][2]=46;
        locate[3][1]=12;
        locate[3][2]=46;
        locate[4][1]=17;
        locate[4][2]=46;
        locate[5][1]=22;
        locate[5][2]=46;
        locate[6][1]=26;
        locate[6][2]=44;
        locate[7][1]=30;
        locate[7][2]=41;
        locate[8][1]=33;
        locate[8][2]=37;
        locate[9][1]=33;
        locate[9][2]=32;
        locate[10][1]=33;
        locate[10][2]=27;
        locate[11][1]=33;
        locate[11][2]=22;

/* Calls function terrain() */
        terrain();

        return;
}

/****************************** terrain() ******************************/
/* This function is WorldToolKit */
terrain()

{
        void my_actions();

/* Creates new universe */
        WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

/* Creates flat terrain */
        terrobj=WTterrain_flat(0.0,35,50,0,0,1.0,1.0,0x0f0,0x0e0,1.0);

/* Defines pose of lights, and create light object */
        at[0]=25;at[1]=-35.0;at[2]=-5.0;
        dir[0]=1.0;dir[1]=1.0;dir[2]=1.0;
        mylight=WTlight_new(at,dir,1.0);

/* Creates the object spaceball */
        spaceball=WTspaceball_new(COM1);

/* Calls function tree() to get tree information */
        tree();
```

```
/* Creates small building, sensor, and target, and places them on the terrain.  Texture
   from "build2.rgb" file is applied to the building */
        bldg=WTobject_newblock(3,-8,3,1,1);
        dp1[X]=29;dp1[Y]=-4;dp1[Z]=29;
        WTobject_translate(bldg,dp1,WTFRAME_WORLD);
        WTobject_settexture(bldg,"build2.rgb",FALSE,FALSE);
        sensor=WTobject_newblock(.5,.5,.5,1,1);
        dp2[X]=0.0;dp2[Y]=-.250;dp2[Z]=0.0;
        WTobject_setposition(sensor,dp2);
        WTobject_changecolor(sensor,0xfff,0xf0f);

/* Loads target from external Autodesk dxf file */
        tank=WTobject_new("tank.dxf",&modelview,0.015,FALSE,FALSE);
        WTobject_changecolor(tank,0xfff,0x777);
        dp3[X]=35.0;dp3[Y]=-1.0;dp3[Z]=0.0;
        rdns=0.0;
        WTobject_setposition(tank,dp3);
        WTobject_rotate(tank,Y,rdns+PI,WTFRAME_WORLD);
        WTobject_setvisibility(tank,FALSE);

/* Prints to screen title headings */
        printf("\nxt\tyt\trange\tprcnt\ttotarea");
        printf("\n___\t___\t______\t______\t_______\n");

/* Sets universe action function to my_actions() */
        WTuniverse_setactions(my_actions);

/* Prepares for simulation to start */
        WTuniverse_ready();

/* Scales spaceball sensitivity to the size of the universe */
        WTsensor_setsensitivity(spaceball, 5.0 *WTuniverse_getradius());

/* Connects the viewpoint to the spaceball */
        WTviewpoint_addsensor(WTuniverse_getviewpoint(),spaceball);

/* Puts some lights on */
        WTlight_setambient(0.3);

/* Enters main simulation loop and starts simulation */
        WTuniverse_go();

/* Deletes universe, simulation loop must be exited to reach this statement */
        WTuniverse_delete();
        return;
}

/************************** my_actions() ***************************/
/* This function determines what actions will occur in the universe */
void my_actions()

{

/* Pressing spaceball button ONE adds an additional window */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON1)

        {

        win1=WTwindow_new(1,500,475,375,WTWINDOW_DEFAULT);
        objview1=WTviewpoint_new();
        p1[X]=xs;p1[Y]=-ho;p1[Z]=ys;
        pp1[X]=0.0;pp1[Y]=0.0;pp1[Z]=0.0;pp1[W]=1.0;
        ppp1[X]=1.0;ppp1[Y]=0.0;ppp1[Z]=1.3;
        WTviewpoint_setposition(objview1,p1);
        WTviewpoint_setorientation(objview1,pp1);
        WTviewpoint_setdirection(objview1,ppp1);
        WTwindow_setviewpoint(win1,objview1);

        }

/* Pressing spaceball button TWO adds a second additional window */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON2)

        {

        win2=WTwindow_new(505,500,500,375,WTWINDOW_DEFAULT);
        objview2=WTviewpoint_new();
        p2[X]=33.0;p2[Y]=-10.0;p2[Z]=52.0;
        pp2[X]=0.0;pp2[Y]=0.0;pp2[Z]=0.0;pp2[W]=1.0;
        ppp2[X]=-1.0;ppp2[Y]=1.0;ppp2[Z]=-1.2;
        WTviewpoint_setposition(objview2,p2);
        WTviewpoint_setorientation(objview2,pp2);
        WTviewpoint_setdirection(objview2,ppp2);
        WTwindow_setviewpoint(win2,objview2);

        }
```

```
/* Pressing spaceball button THREE deletes first window added */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON3)

        {

        WTwindow_delete(win2);

        }

/* Pressing spaceball button FOUR deletes second window added */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON4)

        {

        WTwindow_delete(win1);

        }

/* Pressing spaceball button FIVE makes tank move to next position */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON5)

        {

/* Calls function tgtxtyt() to get position information */
        tgtxtyt();

        pt[X] = xt;pt[Y] = -.75;pt[Z] = yt;

/* Logical section to determine when to rotate tank based on position */
        if (yt == 46)

        {

        rdns = 0.0;

        }

        else if (yt == 44)

        {

        y_rotate = 15;
        rdns = y_rotate * PI / 180;
```

```
}

else if (yt == 41)

{

y_rotate = 25;
rdns = y_rotate * PI / 180;

}

else if (yt == 37)

{

y_rotate = 20;
rdns = y_rotate * PI / 180;

}

else if (yt == 32)

{

y_rotate = 5;
rdns = y_rotate * PI / 180;

}

else if (yt ==27)

{

y_rotate = 0;
rdns = y_rotàte * PI / 180;

}

else if (yt ==22)

{

y_rotate = 0;
rdns = y_rotate * PI / 180;
```

```
        }

/* Creates a tank from external Autodesk dxf file, changes it's color, positions it based
   on the position data, and rotates it if required */
        WTobject_add(tank);
        WTobject_setvisibility(tank,TRUE);
        WTobject_setposition(tank,pt);
        WTobject_rotate(tank,Y,rdns,WTFRAME_WORLD);

/* Calls los() to determine lines of sight to target faces and other calculations */
        los();

/* Calls function datatgt() to display target data */
        datatgt();

/* Zeros out total area and percent data */
        totarea=0;
        prcnt=0.0;

        }

/* Pressing spaceball button EIGHT terminates simulation */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON8)

        {

        WTuniverse_stop();

        }

        return;
}

/****************************** tgtxtyt() *******************************/
/* This function determines the value of xt and yt from array locate[ ][ ] */
tgtxtyt()

{
        c=c+1;
        xt=locate[c][1];
        yt=locate[c][2];

        return;
}
```

```
/******************************* los() ********************************/
/* This function calculates LOS data */
los()

{

/* Begins loop inputting targets moving locations */
                cxt=xt;
                cyt=yt;
                range=sqrt((float)pow((xt-xs),2)+(float)pow((yt-ys),2));

/* Calls the function aspect() to determine how much of the target is presented for
   possible LOS.  Vistgt[ ][ ] is the array returned holding the grid location and
   information on the faces of the target which may be seen, n = the number of
   possible detections and is used for looping the algorithm */
        aspect();

/* Loops to check possible LOS for all surfaces presented by target */
        r=0;

        for (j=1; j<=n; j++)

                {

/* Zeros atten for each run and gets target grid & height data for stepping LOS */
                atten=0;
                attenf=0;
                xt=vistgt[j][1];
                yt=vistgt[j][2];
                ht=vistgt[j][3];

/* Calculates target and observer heights. First ground height must be found by
   subtracting vegetation height from absolute height.  Then sensor and target heights
   above ground are added to obtain absolute elevations of sensor and target */
                zs=elev[xs*50+ys]-vgh[xs*50+ys];
                zt=elev[xt*50+yt]-vgh[xt*50+yt];
                zs=zs+ho;
                zt=zt+ht;

/* Determines difference btwn x & y coordinates, and converts from integer to real */
                rdx=(xt-xs);
                rdy=(yt-ys);
                idx=rdx;
                idy=rdy;
```

```
/* If idy > idx skip to stepping in y direction, else proceed stepping in x direction */
        if (idy >= idx)

                {

                goto YSTEP;

                }
        else
                {

                y=ys;
                dy=(yt-ys)/rdx;
                z=zs;
                dz=(zt-zs)/rdx;

/* This if-else statement ensures moving from sensor to target.  Move only from
   sensor because dist from sensor will affect attenuation level of any obstructions */
        if (xt > xs)

                {

/* Apply slope to each step to determine grid passing thru and LOS height in that grid
   compare height to ground height, no LOS exists if ground height > LOS height:
   z            = height of LOS
   ztree        = height of vegetation
   zdirt        = height of the ground */
                istart=xs+1;
                istop=xt;
                ystart=y;

        for (ix=istart; ix<=istop; ix++)

                {

                y=y+dy;
                z=z+dz;
                ninty();
                ztree=elev[ix*50+iy];
                zdirt=ztree-vgh[ix*50+iy];

/* Calculate dist from sensor to grid where LOS currently in heading toward  target */
                dist=sqrt((float)pow((istart-ix),2)+(float)pow((ystart-iy),2));
```

```
/* Calls function compare() */
        compare();

        if (nolos != 0)

                {

                goto FINISH;

                }

                }

        goto CONTINUE;

                }
        else
                {

                istart=xs-1;
                istop=xt;
                ystart=y;

        for (ix=istart; ix<=istop; ix--)

                {

                y=y+dy;
                z=z+dz;
                ninty();
                ztree=elev[ix*50+iy];
                zdirt=ztree-vgh[ix*50+iy];
                dist=sqrt((float)pow((istart-ix),2)+(float)pow((ystart-iy),2));

/* Calls function compare() */
        compare();

        if (nolos != 0)

                {

                goto FINISH;

                }
```

```
            }

        goto CONTINUE;

            }

            }

/* This section is used if stepping in y direction and same as stepping in x direction */
        YSTEP:

            x=xs;
            dx=(xt-xs)/rdy;
            z=zs;
            dz=(zt-zs)/rdy;

        if (yt > ys)

            {

            istart=ys+1;
            istop=yt;
            xstart=x;

        for (iy=istart; iy<=istop; iy++)

            {

            x=x+dx;
            z=z+dz;
            nintx();
            ztree=elev[ix*50+iy];
            zdirt=ztree-vgh[ix*50+iy];
            dist=sqrt((float)pow((xstart-ix),2)+(float)pow((istart-iy),2));

/* Calls function compare() */
        compare();

        if (nolos != 0)

            {

            goto FINISH;
            }

            }

    goto CONTINUE;

            }
    else
            {

            istart=ys-1;
            istop=yt;
            xstart=x;

    for (iy=istart; iy<=istop; iy--)

            {

            x=x+dx;
            z=z+dz;
            nintx();
            ztree=elev[ix*50+iy];
            zdirt=ztree-vgh[ix*50+iy];
            dist=sqrt((float)pow((xstart-ix),2)+(float)pow((istart-iy),2));

/* Calls function compare() */
    compare();

    if (nolos != 0)

            {

            goto FINISH;

            }

            }

    goto CONTINUE;

            }

    CONTINUE:
```

```
/* At this point LOS has atten = 0 or a number.  Apply this attenuation factor to proj
   area of target to reduce area of visibility.  Proj area is determined based on surface
   direction of face in question.  This value is 1 for a vertical surface and 2 for a
   horizontal surface.  This value is stored in vistgt[ ][ ] array which describes target */
      if (vistgt[j][4] == 0)

            {

            aproj=rdy/sqrt(pow(rdy,2)+pow(rdx,2));

            }
      else
            {

            aproj=rdx/sqrt(pow(rdy,2)+pow(rdx,2));

            }

/* Sees if there is any attenuation to be applied, if so apply it */
      if (atten == 0)

            {

            visaproj=aproj;

            }
      else
            {

            visaproj=(1-atten)*aproj;

            }

/* Sums total visible area presented by target */
            totarea=totarea+visaproj;

/* Zeros aproj and visaproj for next LOS */
            aproj=0;
            visaproj=0;
            r=r+1;

      FINISH:

            nolos=0;
```

```
                }

                return;
}

/****************************** aspect() ******************************/
/* This subroutine assigns target grid locations based on central input location.  It then
   uses sensor and target locations to determine which faces of target are ideally visible
   to sensor.  Faces which are visible have there information stored in array vistgt[ ][ ]
   for return to main program.  The number of faces ideally visible are also returned.
   Assign target data to array tgt[ ][ ] based on center grid of target 3x3 each
   exterior vertical face of target is represented */
aspect()

{

        tgt[1][1]=xt-1;
        tgt[1][2]=yt-2;
        tgt[1][4]=0;
        tgt[2][1]=xt-2;
        tgt[2][2]=yt-1;
        tgt[2][4]=1;
        tgt[3][1]=xt-2;
        tgt[3][2]=yt;
        tgt[3][4]=1;
        tgt[4][1]=xt-2;
        tgt[4][2]=yt+1;
        tgt[4][4]=1;
        tgt[5][1]=xt-1;
        tgt[5][2]=yt+2;
        tgt[5][4]=0;
        tgt[6][1]=xt;
        tgt[6][2]=yt+2;
        tgt[6][4]=0;
        tgt[7][1]=xt+1;
        tgt[7][2]=yt+2;
        tgt[7][4]=0;
        tgt[8][1]=xt+2;
        tgt[8][2]=yt+1;
        tgt[8][4]=1;
        tgt[9][1]=xt+2;
        tgt[9][2]=yt;
        tgt[9][4]=1;
        tgt[10][1]=xt+2;
        tgt[10][2]=yt-1;
        tgt[10][4]=1;
        tgt[11][1]=xt+1;
        tgt[11][2]=yt-2;
        tgt[11][4]=0;
        tgt[12][1]=xt;
        tgt[12][2]=yt-2;
        tgt[12][4]=0;
        tgt[13][1]=xt;
        tgt[13][2]=yt-1;
        tgt[13][4]=0;
        tgt[14][1]=xt-1;
        tgt[14][2]=yt;
        tgt[14][4]=1;
        tgt[15][1]=xt;
        tgt[15][2]=yt+1;
        tgt[15][4]=0;
        tgt[16][1]=xt+1;
        tgt[16][2]=yt;
        tgt[16][4]=1;

/* Assigns target heights, in tgt[ ][ ] */
        for (j=1; j<=12; j++)

                {

                tgt[j][3]=1;

                }

        for (j=13; j<=16; j++)

                {

                tgt[j][3]=2;

                }

/* Establishes bounds of the target */
                xmax=xt+1;
                xmin=xt-1;
                ymax=yt+1;
                ymin=yt-1;
```

```
/* Determines visible sectors of target */
        if ((xs <= xmax) && (xs >= xmin))

                {

        if (ys > ymax)

                {

/* Sensor is directly above target grids, upper faces */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[5][j];
                vistgt[2][j]=tgt[6][j];
                vistgt[3][j]=tgt[7][j];
                vistgt[4][j]=tgt[15][j];
                n=4;

/* Allows for skewed view of a top block side */
        if (xs < xt)

                {

                vistgt[5][j]=tgt[14][j];
                n=5;

                }

        if (xs > xt)

                {

                vistgt[5][j]=tgt[16][j];
                n=5;

                }

                }

        goto END;

                }
        else
                {

/* If the sensor is vertically in line with target grids and it is not above target it must
   be below it, therefore it sees lower faces */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[1][j];
                vistgt[2][j]=tgt[12][j];
                vistgt[3][j]=tgt[11][j];
                vistgt[4][j]=tgt[13][j];
                n=4;

/* Allows for skewed view of a top block side */
        if (xs < xt)

                {

                vistgt[5][j]=tgt[14][j];
                n=5;

                }

        if (xs > xt)

                {

                vistgt[5][j]=tgt[16][j];
                n=5;

                }

                }

        goto END;

                }

                }

        if (xs < xmin)
```

```
                {

        if ((ys >= ymin) && (ys <= ymax))

                {

/* Sensor is horizontally aligned with target grids and to left of target, therefore it sees
   left side faces */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[2][j];
                vistgt[2][j]=tgt[3][j];
                vistgt[3][j]=tgt[4][j];
                vistgt[4][j]=tgt[14][j];
                n=4;

/* Allows for skewed view of top block side */
        if (ys < yt)

                {

                vistgt[5][j]=tgt[13][j];
                n=5;

                }

        if (ys > yt)

                {

                vistgt[5][j]=tgt[15][j];
                n=5;

                }

                }

        goto END;

                }

        if (ys < ymin)
```

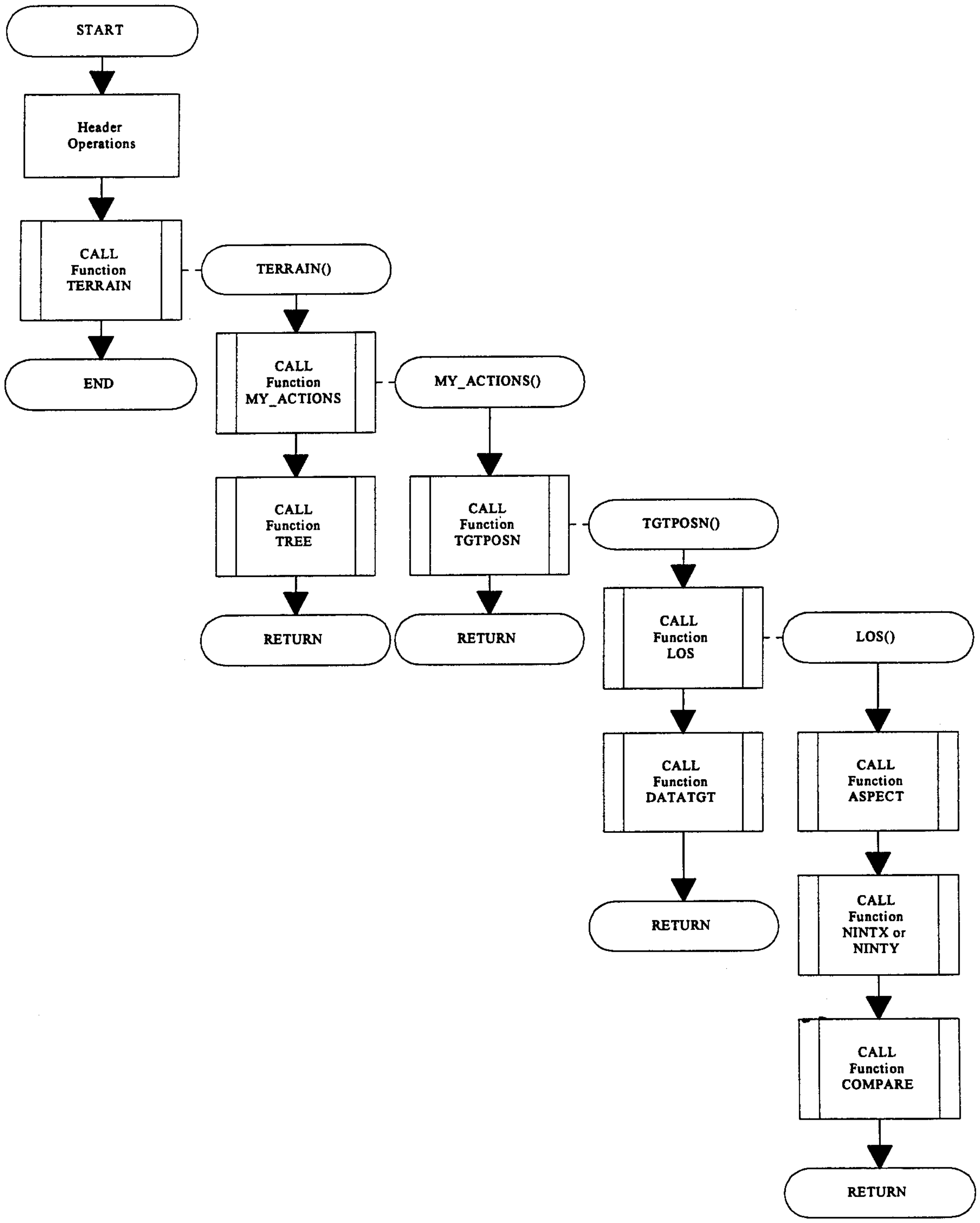
```
                {

/* Sensor is to left and below target, all of left side and lower faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[1][j];
                vistgt[2][j]=tgt[2][j];
                vistgt[3][j]=tgt[3][j];
                vistgt[4][j]=tgt[4][j];
                vistgt[5][j]=tgt[12][j];
                vistgt[6][j]=tgt[11][j];
                vistgt[7][j]=tgt[13][j];
                vistgt[8][j]=tgt[14][j];
                n=8;

                }

        goto END;

                }

        if (ys > ymax)

                {

/* Sensor is to left side and above target, all of left side and upper faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[2][j];
                vistgt[2][j]=tgt[3][j];
                vistgt[3][j]=tgt[4][j];
                vistgt[4][j]=tgt[5][j];
                vistgt[5][j]=tgt[5][j];
                vistgt[6][j]=tgt[7][j];
                vistgt[7][j]=tgt[14][j];
                vistgt[8][j]=tgt[15][j];
                n=8;

                }
        goto END;

                }

                }

/* Sensor is right of target */
        if ((ys >= ymin) && (ys <= ymax))

                {

/* Sensor horizontally aligned w/target and right of target, therefore right faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[8][j];
                vistgt[2][j]=tgt[9][j];
                vistgt[3][j]=tgt[10][j];
                vistgt[4][j]=tgt[16][j];
                n=4;

/* Allows a skewed view of a top side */
        if (ys < yt)

                {

                vistgt[5][j]=tgt[13][j];
                n=5;

                }

        if (ys > yt)

                {

                vistgt[5][j]=tgt[15][j];
                n=5;

                }

                }

        goto END;
```

```
                }

        if (ys < ymin)

                {

  /* Sensor is right of and below target, all right and lower faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[8][j];
                vistgt[2][j]=tgt[9][j];
                vistgt[3][j]=tgt[10][j];
                vistgt[4][j]=tgt[1][j];
                vistgt[5][j]=tgt[12][j];
                vistgt[6][j]=tgt[11][j];
                vistgt[7][j]=tgt[13][j];
                vistgt[8][j]=tgt[16][j];
                n=8;

                }
        goto END;
                }

        if (ys > ymax)

                {

  /* Sensor is right and above target, all right and upper faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[8][j];
                vistgt[2][j]=tgt[9][j];
                vistgt[3][j]=tgt[10][j];
                vistgt[4][j]=tgt[5][j];
                vistgt[5][j]=tgt[6][j];
                vistgt[6][j]=tgt[7][j];
                vistgt[7][j]=tgt[16][j];
                vistgt[8][j]=tgt[15][j];
                n=8;
                }

        goto END;

                }

        END:

        return;
}

/****************************** nintx() ******************************/
/* This function round up or down to the nearest integer for ix */
nintx()

{
        ix=floor(x);
        zx=fabs(x-ix);

        if (zx >= .5)

                {

                ix=ceil(x);

                }
        else
                {

                ix=floor(x);

                }

        return;
}

/****************************** ninty() ******************************/
/* This function round up or down to the nearest integer for iy */
ninty()

{
        iy=floor(y);
        zy=fabs(y-iy);
        if (zy >= .5)

                {

                iy=ceil(y);

                }
        else
                {

                iy=floor(y);

                }

        return;
}

/***************************** compare() *****************************/
/* This function compares LOS data to terrain data to see if LOS is obstructed */
compare()

{

/* Compares LOS height to ground height */
        if (z < zdirt)

                {

                nolos=1;

                }

        else if (z < ztree)

                {

/* The following determine attenuation due to vegetation.  If feature is 200 m away
   then appears as solid object, distance arbitrarily selected.  Checks UCI if object has
   height but no UCI assume to be trunk of object/man made structure, LOS blocked */
        if (uci[ix*50+iy] == 0)

                {

                nolos=2;
                }

        else if (z > uci[ix*50+iy])

                {

/* Checks nature bit, structures block LOS. Manmade objects have a nature bit of 0 */
        if (nat[ix*50+iy] == 0)

                {

                nolos=3;

                }

/* If program passes above test then obstruction is vegetation and any other parts of
   tree will be assumed as foliage.  Current assumption is foliage of 1 meter thickness
   has an attenuation of 30%.  This value is modified as function of distance until
   modified value reaches an attenuation of 100% at terminal distance, 200 meters.  At
   200 meters all objects appear solid.  It is assumed the modification factor is linear */
        if (dist > 200)

                {

                nolos=4;

                }
        else
                {

                attenf=denfol*(1+2.33*dist/200);

                }

                }

/* Allows for sensor hiding behind or in foliage to see through w/out attenuation */
        if (dist <= 1)

                {

                attenf=0;

                }
```

```
/* Sums attenuations */
                atten=atten+attenf;

/* If total attenuation exceeds 95%, LOS is blocked */
        if (atten > .95)

                {

                nolos=5;

                }

                }

        return;
}

/**************************** datatgt() ******************************/
/* This function displays to screen:  range, % of target faces visible, and total area of
   target faces visible */
datatgt()

{
        prcnt=floor((r / n)*100);
        printf("\n%d\t%d\t%3.2f\t%d\t%3.2f\n",cxt,cyt,range,prcnt,totarea);

        return;
}

/****************************** tree() *******************************/
/* This function creates trees & places them on terrain from info in "tree.dat" */
static void tree()

{

/* Declare variables */
        int row;
        float rad,x,z;

/* Opens file "tree.dat" and reads dimensional data */
        fp1=fopen("tree.dat","r");
        for (row=0; row<7; row++)

        {
```

```
        fscanf(fp1," %f %f %f\n",&mdata[row][0],&mdata[row][1],&mdata[row][2]);

/* Defines variables rad, x, z */
        rad=mdata[row][0];x=mdata[row][1];z=mdata[row][2];

/* Defines cube as new sphere and cyl as a new cylinder attaches them together.
   Texture from "grass.rgb" and "wood.rgb" is applied to the tree trunk and leaves */
        factors[X]=factors[Z]=.50;factors[Y]=1.25;
        sphere = WTobject_newsphere(rad,5,5,1,1,1);
        WTobject_stretch(sphere,factors,p,WTFRAME_WORLD);
        cyl = WTobject_newcylinder(rad/1.25,rad/5,4,1,1,1);
        dp4[X]=0;dp4[Y]=-(1.3*rad);dp4[Z]=0;
        WTobject_translate(sphere,dp4,WTFRAME_WORLD);
        WTobject_attach(sphere,cyl);
        WTobject_add(sphere);
        dp5[X]=x;dp5[Y]=-rad/4;dp5[Z]=z;
        WTobject_translate(sphere,dp5,WTFRAME_WORLD);
        WTobject_settexture(sphere,"grass.rgb",FALSE,FALSE);
        WTobject_settexture(cyl,"wood.rgb",FALSE,FALSE);

        }

        fclose(fp1);
        return;
}
```

APPENDIX H. FLOWCHART PROG2.C



APPENDIX I. PROGRAM PROG2.C

```
/* PROGRAM PROG2.C */
/*************************************************************************/
/* This program calculates Line Of Sight (LOS) data and visualizes a target moving*/
/* through a 35x50 grid using a database of Pegasus numbers and is designed to    */
/* incorporate attenuation by vegetation. The main section of this program calls  */
/* function terrain() which is used to start WorldToolKit and create a universe   */
/* where there is a flat terrain with trees and a building located on it. The trees and*/
/* building are represented as close approximations to real objects. The simulation */
/*  has the target moving along a path behind the trees and buildings. Function   */
/* terrain() calls function los() which is used to calculate target data.         */
/*************************************************************************/
/* Standard C and WorldToolKit header files */
#include <stdio.h>
#include <math.h>
#include"wt.h"
#include"wt.p"

/* Defines a pointer spaceball to structure type WTsensor */
static WTsensor *spaceball=NULL;
static WTsensor *mouse=NULL;

/* Calls function tree() */
static void tree();

/* Defines windows, viewpoints, nodes, and paths */
static WTwindow *win1, *win2, *win3;
static WTviewpoint *objview1, *objview2, *objview3;

/* Defines objects */
WTobject *bldg, *copy, *cyl, *original, *sensor, *sphere, *tank, *terrobj;
WTobject *nodeobj;

/* Defines path */
static WTpath *tnkpth;

/* Defines pointers to WTp3, WTq, and WTpq structures */
WTp3 at,dp1,dp2,dp3,dp4,dp5,dir,factors,p,p1,p2,ppp1,ppp2,pp,pt;
WTq pp1,pp2;
WTpq modelview;

/* Defines light object to be a pointer to type WTlight */
WTlight *mylight;
```

```
/* Initialize arrays and declares variables */
int ans,c,cxt,cyt,ho,ht,i,ix,iy,idx,idy,istart,istop,j,n,nmbr,nolos,prcnt,win,wdow,xs,xt;
int xmax,xmin,xstart,ys,yt,ymax,ymin,ystart,zs,zt,zdirt,ztree,binelev[1750],binnat[1750];
int binuci[1750],binvghindex[1750],binvid[1750],elev[1750],nat[1750],realvgh[10];
int tgt[17][5],tile[1750]uci[1750],vgh[1750],vid[1750],vghindex[1750],vistgt[9][9];
float aproj,atten,attenf,dx,dy,dz,denfol,dist,r,rad,rdx,rdy,row,rdns,range;
float totarea,visaproj,x,y,y_rotate,z,zx,z,mdata[7][3];

/* Standard C file pointers */
FILE *fp;
FILE *fp1;

main()

{

/* Reads database information into array from data file "btlfld_terr.dat" */
        fp=fopen("btlfld_terr.dat","r");
        for (i=0; i<=1749; i++)

                {

                fscanf(fp," %d", &tile[i]);

                }

        fclose(fp);

/* Assigns 1 meter of foliage an attenuation of 30% of the LOS */
        denfol=0.3;

/* The vegetation height from pegasus has values of 0-15, each of these values
   represents a particular height or range of heights. The following is used to correlate
   the vgh value to a meaningful height */
        realvgh[0]=0;
        realvgh[1]=0;
        realvgh[2]=1;
        realvgh[3]=2;
        realvgh[4]=3;
        realvgh[5]=4;
        realvgh[6]=5;
        realvgh[7]=8;
        realvgh[8]=10;
        realvgh[9]=15;
        realvgh[10]=20;
        realvgh[11]=25;
        realvgh[12]=30;
        realvgh[13]=35;
        realvgh[14]=40;
        realvgh[15]=47;

/* Once the database for the grid desired has been read into an array, the components
   of information are extracted from the 32 bit number using the AND and SHIFT
   functions. The groups of information of use in the program are placed in their own
   arrays for rapid recall during calculations as follows:
   elevation of highest point in grid       - ELEV
   under cover index                        - UCI
   converted vegetation height              - VGH
   vegetation ID                            - VID
   nature bit                               - NAT */
              for (i=0; i<=1749; i++)

              {

              binuci[i]=tile[i] & 786432;
              binvghindex[i]=tile[i] & 15360;
              binvid[i]=tile[i] & 768;
              binnat[i]=tile[i] & 128;
              binelev[i]=tile[i] & 4292870144;
              uci[i] = binuci[i] >> 18;
              vghindex[i] = binvghindex[i] >> 10;
              vgh[i]=realvgh[vghindex[i]];
              vid[i] = binvid[i] >> 8;
              nat[i] = binnat[i] >> 7;
              elev[i] = binelev[i] >> 21;

              }

/* This program is currently written to run in a stand alone mode.   Sensor information
   has a default value, but can be changed by answering prompts from the keyboard.
   Accepts default values or input sensor location and height */
        printf("\nThe Default Sensor Coordinates are xs = 0, zs = 0");
        printf("\nand hs = 1\n");
        printf("\nDo you wish to Change these values?  1=Yes, 2=No\n");
        scanf(" %d",&ans);

        if (ans == 1)
```

```
                {

                BEGIN:

                printf("\nEnter Sensor X Coordinate (X,Y)\n");
                scanf("%d,%d",&xs,&ys);
                printf("\nEnter Sensor Height\n");
                scanf("%d",&ho);
                printf("\nAre You Sure?  1=Yes  2=No \n");
                scanf(" %d", &ans);

        if (ans == 2)

                {

                goto BEGIN;

                }

                }
        else
                {

                xs=0;
                ys=0;
                ho=1;

                }

        printf("\nPress Spaceball Button 1 to add a window\n");
        printf("\nPress Spaceball Button 2 to add another window\n");
        printf("\nPress Spaceball Button 3 to delete first window added\n");
        printf("\nPress Spaceball Button 4 to delete second window added\n");
        printf("\nPress Spaceball Button 5 to start tank motion\n");
        printf("\nPress Spaceball Button 6 to replay simulation\n");
        printf("\nPress Spaceball Button 8 to end simulation\n");

/* Calls function terrain() */
        terrain();

return;

}
```

```
/****************************** terrain() ******************************/
/* This function is WorldToolKit */
terrain()

{
        void my_actions();

/* Creates new universe */
        WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

/* Creates the object terrobj */
        terrobj=WTterrain_flat(0.0,35,50,0,0,1.0,1.0,0x0f0,0x0e0,1.0);

/* Defines pose of lights, and create light object */
        at[0]=25;at[1]=-35.0;at[2]=-5.0;
        dir[0]=1.0;dir[1]=1.0;dir[2]=1.0;
        mylight=WTlight_new(at,dir,1.0);

/* Creates the object spaceball or mouse */
        spaceball=WTspaceball_new(COM1);

/* Calls function tree() to get tree information */
        tree();

/* Creates small building, sensor, and target, and places them on the terrain */
        bldg=WTobject_newblock(3,-8,3,1,1);
        dp1[X]=29;dp1[Y]=-4;dp1[Z]=29;
        WTobject_translate(bldg,dp1,WTFRAME_WORLD);
        sensor=WTobject_newblock(.5,.5,.5,1,1);
        dp2[X]=0.0;dp2[Y]=-.250;dp2[Z]=0.0;
        WTobject_setposition(sensor,dp2);
        WTobject_changecolor(sensor,0xfff,0xf0f);

/* Loads tank from external Autodesk dxf file */
        tank=WTobject_new("tank180.dxf",&modelview,0.015,FALSE,FALSE);
        WTobject_changecolor(tank,0xfff,0x777);
        dp3[X]=15.0;dp3[Y]=5.0;dp3[Z]=25.0;
        rdns=0.0;
        WTobject_setposition(tank,dp3);
        WTobject_rotate(tank,Y,rdns+PI,WTFRAME_WORLD);

/* Tank path is loaded from external file */
        nodeobj=WTobject_newblock(.1,.1,.1,1,1);
        tnkpth=WTpath_load("tank.pth",nodeobj);
```

```
/* Prints to screen title headings */
        printf("\nxt\tyt\trange\tprcnt\ttotarea");
        printf("\n__\t__\t_____\t_____\t_______\n");

/* Sets universe action function to my_actions() */
        WTuniverse_setactions(my_actions);

/* Prepares for simulation to start */
        WTuniverse_ready();

/* Scales spaceball or mouse sensitivity to the size of the universe */
        WTsensor_setsensitivity(spaceball, 5.0 *WTuniverse_getradius());

/* Connects the viewpoint to the spaceball or mouse */
        WTviewpoint_addsensor(WTuniverse_getviewpoint(),spaceball);

/* Puts some lights on */
        WTlight_setambient(0.3);

/* Enters main simulation loop and starts simulation */
        WTuniverse_go();

/* Deletes universe, simulation loop must be exited to reach this statement */
        WTuniverse_delete();

        return;
}

/************************** my_actions() ***************************/
/* This function determines what actions will occur in the universe */
void my_actions()

{

/* Pressing spaceball button ONE adds an additional window */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON1)

        {

        win1=WTwindow_new(1,500,475,375,WTWINDOW_DEFAULT);
        objview1=WTviewpoint_new();
        p1[X]=xs;p1[Y]=-ho;p1[Z]=ys;
        pp1[X]=0.0;pp1[Y]=0.0;pp1[Z]=0.0;pp1[W]=1.0;
        ppp1[X]=1.0;ppp1[Y]=0.0;ppp1[Z]=1.3;
```

```
         WTviewpoint_setposition(objview1,p1);
         WTviewpoint_setorientation(objview1,pp1);
         WTviewpoint_setdirection(objview1,ppp1);
        WTwindow_setviewpoint(win1,objview1);

         }

/* Pressing spaceball button TWO adds a second additional window */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON2)

        {

        win2=WTwindow_new(505,500,500,375,WTWINDOW_DEFAULT);
        objview2=WTviewpoint_new();
        p2[X]=33.0;p2[Y]=-10.0;p2[Z]=52.0;
        pp2[X]=0.0;pp2[Y]=0.0;pp2[Z]=0.0;pp2[W]=1.0;
        ppp2[X]=-1.0;ppp2[Y]=1.0;ppp2[Z]=-1.2;
        WTviewpoint_setposition(objview2,p2);
       WTviewpoint_setorientation(objview2,pp2);
       WTviewpoint_setdirection(objview2,ppp2);
       WTwindow_setviewpoint(win2,objview2);

        }

/* Pressing spaceball button THREE deletes first window added */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON3)

        {

        WTwindow_delete(win2);

        }

/* Pressing spaceball button FOUR deletes second window added */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON4)

        {

        WTwindow_delete(win1);

        }

/* Pressing spaceball button FIVE starts tank motion along path */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON5)
        {

        WTpath_setobject(tnkpth,tank);
        WTpath_play(tnkpth);
        WTpath_record(tnkpth);

        }

/* Pressing spaceball button SIX restarts tank motion */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON6)

        {

        WTpath_stop(tnkpth);
        WTpath_rewind(tnkpth);
        WTpath_play(tnkpth);

        }

/* Based on tank position, function tgtposn() called */
        WTobject_getposition(tank,pp);

        if (pp[X] == 2)

        {

        tgtposn();

        }

        if (pp[X] == 7)

        {

        tgtposn();

        }

        if (pp[X] == 12)

        {

        tgtposn();
}
if (pp[X] == 17)
{
tgtposn();
}
if (pp[X] == 22)
{
tgtposn();
}
if (pp[X] == 26)
{
tgtposn();
}
if (pp[X] == 30)
{
tgtposn();
}
if ((pp[X] == 33) && (pp[Z] == 37))
{
tgtposn();
}
if ((pp[X] == 33) && (pp[Z] == 32))
```

```
        {

        tgtposn();

        }

        if ((pp[X] == 33) && (pp[Z] == 27))

        {

        tgtposn();

        }

        if ((pp[X] == 33) && (pp[Z] == 22))

        {

        tgtposn();

        }

/* Pressing spaceball button EIGHT terminates simulation */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON8)

        {

        WTuniverse_stop();

        }

        return;
}

/**************************** tgtposn() ******************************/
/* This function calls los() and datatgt(), receives value of xt and zt from
   WTobject_getposition, and displays target data to screen */
tgtposn()

{

        xt=pp[X];yt=pp[Z];
```

```
/* Calls function los() to determine LOS to target faces and other calculations */
        los();

/* Calls function datatgt() to display target data */
        datatgt();

/* Zeros out total area and percent data */
        totarea=0;
        prcnt=0.0;

        return;
}

/******************************* los() *******************************/
/* This function calculates LOS data */
los()

{

/* Begins loop inputting targets moving locations */
                cxt=xt;
                cyt=yt;
                range=sqrt((float)pow((xt-xs),2)+(float)pow((yt-ys),2));

/* Calls the function aspect() to determine how much of the target is presented for
   possible LOS.  Vistgt[ ][ ] is the array returned holding the grid location and
   information on the faces of the target which may be seen, n = the number of
   possible detections and is used for looping the algorithm */
                aspect();

/* Loops to check possible LOS for all surfaces presented by target */
                r=0;

        for (j=1; j<=n; j++)

                {

/* Zeros atten for each run and gets target grid height data for stepping LOS */
                atten=0;
                attenf=0;
                xt=vistgt[j][1];
                yt=vistgt[j][2];
                ht=vistgt[j][3];
```

```
/* Calculates target and observer heights. First ground height must be found by
   subtracting vegetation height from absolute height.  Then sensor and target heights
   above ground are added to obtain absolute elevations of sensor and target */
            zs=elev[xs*50+ys]-vgh[xs*50+ys];
            zt=elev[xt*50+yt]-vgh[xt*50+yt];
            zs=zs+ho;
            zt=zt+ht;

/* Determines difference btwn x & y coordinates and convert from integer real */
            rdx=(xt-xs);
            rdy=(yt-ys);
            idx=rdx;
            idy=rdy;

/* If idy > idx skip to stepping in y direction, else proceed in x direction */
        if (idy >= idx)

            {

            goto YSTEP;

            }
        else
            {

            y=ys;
            dy=(yt-ys)/rdx;
            z=zs;
            dz=(zt-zs)/rdx;

/* This if-else  statement ensures moving from sensor to target.  Move only from
   sensor because dist from sensor will affect attenuation level of any obstructions */
        if (xt > xs)

            {

/* Apply slope to each step to determine grid passing thru and LOS height in that grid
   compare height to ground height, no LOS will exist if ground height > LOS height:
   z        = height of LOS
   ztree    = height of vegetation
   zdirt    = height of the ground */
            istart=xs+1;
            istop=xt;
            ystart=y;
```

```
    for (ix=istart; ix<=istop; ix++)

            {

            y=y+dy;
            z=z+dz;
            ninty();
            ztree=elev[ix*50+iy];
            zdirt=ztree-vgh[ix*50+iy];

/* Calculate dist from sensor to grid where LOS currently in heading toward target */
            dist=sqrt((float)pow((istart-ix),2)+(float)pow((ystart-iy),2));

/* Calls function compare() */
        compare();

        if (nolos != 0)

            {

            goto FINISH;

            }

            }

        goto CONTINUE;

            }
        else
            {

            istart=xs-1;
            istop=xt;
            ystart=y;

        for (ix=istart; ix<=istop; ix--)

            {

            y=y+dy;
            z=z+dz;
            ninty();
            ztree=elev[ix*50+iy];
```

```
                zdirt=ztree-vgh[ix*50+iy];
                dist=sqrt((float)pow((istart-ix),2)+(float)pow((ystart-iy),2));

/* Calls function compare() */
        compare();

        if (nolos != 0)

                {

                goto FINISH;

                }

                }

        goto CONTINUE;

                }

                }

/* This section is used if stepping in y direction and same as stepping in x direction */
        YSTEP:

                x=xs;
                dx=(xt-xs)/rdy;
                z=zs;
                dz=(zt-zs)/rdy;

        if (yt > ys)

                {

                istart=ys+1;
                istop=yt;
                xstart=x;

        for (iy=istart; iy<=istop; iy++)

                {

                x=x+dx;
                z=z+dz;
                nintx();
                ztree=elev[ix*50+iy];
                zdirt=ztree-vgh[ix*50+iy];
                dist=sqrt((float)pow((xstart-ix),2)+(float)pow((istart-iy),2));

/* Calls function compare() */
        compare();

        if (nolos != 0)

                {

                goto FINISH;

                }

                }

        goto CONTINUE;

                }
        else
                {

                istart=ys-1;
                istop=yt;
                xstart=x;

        for (iy=istart; iy<=istop; iy--)

                {

                x=x+dx;
                z=z+dz;
                nintx();
                ztree=elev[ix*50+iy];
                zdirt=ztree-vgh[ix*50+iy];
                dist=sqrt((float)pow((xstart-ix),2)+(float)pow((istart-iy),2));

/* Calls function compare() */
        compare();

        if (nolos != 0)
```

```
                {

                goto FINISH;

                }

                }

        goto CONTINUE;

                }

        CONTINUE:

/* At this point LOS has atten = 0 or a number.  Apply this attenuation factor to proj
   area of target to reduce area of visibility.  Proj area is determined based on surface
   direction of face in question.  This value is 1 for a vertical surface and 2 for a
   horizontal surface.  This value is stored in array vistgt[ ][ ] which describes target */
        if (vistgt[j][4] == 0)

                {

                aproj=rdy/sqrt(pow(rdy,2)+pow(rdx,2));

                }

        else
                {

                aproj=rdx/sqrt(pow(rdy,2)+pow(rdx,2));

                }

/* Sees if there is any attenuation to be applied, if so apply it */
        if (atten == 0)

                {

                visaproj=aproj;

                }
        else
                {
```

```
                visaproj=(1-atten)*aproj;

                }

/* Sums total visible area presented by target */
                totarea=totarea+visaproj;

/* Zeros aproj and visaproj for next LOS */

                aproj=0;
                visaproj=0;
                r=r+1;

        FINISH:

                nolos=0;

                }

                return;
}

/***************************** aspect() *****************************/
/* This subroutine assigns target grid locations based on central input location.  It then
   uses sensor and target locations to determine which faces of target are ideally visible
   to sensor.  Faces which are visible have there information stored in array vistgt[ ][ ]
   for return to main program.  The number of faces ideally visible are also returned.
   Assign target data to array tgt[ ][ ] based on center grid of target 3x3 each exterior
   vertical face of target is represented */
aspect()

{

        tgt[1][1]=xt-1;
        tgt[1][2]=yt-2;
        tgt[1][4]=0;
        tgt[2][1]=xt-2;
        tgt[2][2]=yt-1;
        tgt[2][4]=1;
        tgt[3][1]=xt-2;
        tgt[3][2]=yt;
        tgt[3][4]=1;
        tgt[4][1]=xt-2;
        tgt[4][2]=yt+1;
        tgt[4][4]=1;
        tgt[5][1]=xt-1;
        tgt[5][2]=yt+2;
        tgt[5][4]=0;
        tgt[6][1]=xt;
        tgt[6][2]=yt+2;
        tgt[6][4]=0;
        tgt[7][1]=xt+1;
        tgt[7][2]=yt+2;
        tgt[7][4]=0;
        tgt[8][1]=xt+2;
        tgt[8][2]=yt+1;
        tgt[8][4]=1;
        tgt[9][1]=xt+2;
        tgt[9][2]=yt;
        tgt[9][4]=1;
        tgt[10][1]=xt+2;
        tgt[10][2]=yt-1;
        tgt[10][4]=1;
        tgt[11][1]=xt+1;
        tgt[11][2]=yt-2;
        tgt[11][4]=0;
        tgt[12][1]=xt;
        tgt[12][2]=yt-2;
        tgt[12][4]=0;
        tgt[13][1]=xt;
        tgt[13][2]=yt-1;
        tgt[13][4]=0;
        tgt[14][1]=xt-1;
        tgt[14][2]=yt;
        tgt[14][4]=1;
        tgt[15][1]=xt;
        tgt[15][2]=yt+1;
        tgt[15][4]=0;
        tgt[16][1]=xt+1;
        tgt[16][2]=yt;
        tgt[16][4]=1;

/* Assigns target heights, in tgt[ ][ ] */
        for (j=1; j<=12; j++)

                {

                tgt[j][3]=1;
                }

        for (j=13; j<=16; j++)

                {

                tgt[j][3]=2;

                }

/* Establishs bounds of the target */
                xmax=xt+1;
                xmin=xt-1;
                ymax=yt+1;
                ymin=yt-1;

/* Determine visible sectors of target */
        if ((xs <= xmax) && (xs >= xmin))

                {

        if (ys > ymax)

                {

/* Sensor is directly above target grids, upper faces */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[5][j];
                vistgt[2][j]=tgt[6][j];
                vistgt[3][j]=tgt[7][j];
                vistgt[4][j]=tgt[15][j];
                n=4;

/* Allows for skewed view of a top block side */
        if (xs < xt)

                {

                vistgt[5][j]=tgt[14][j];
                n=5;
```

```
            }

     if (xs > xt)

            {

            vistgt[5][j]=tgt[16][j];
            n=5;

            }

            }

     goto END;

            }
     else
            {

/* If the sensor is vertically in line with target grids and it is not above target it must
   be below it, therefore it sees lower faces */

     for (j=1; j<=4; j++)

            {

            vistgt[1][j]=tgt[1][j];
            vistgt[2][j]=tgt[12][j];
            vistgt[3][j]=tgt[11][j];
            vistgt[4][j]=tgt[13][j];
            n=4;

/* Allows for skewed view of a top block side */
     if (xs < xt)

            {

            vistgt[5][j]=tgt[14][j];
            n=5;

            }

     if (xs > xt)
```

```
            {

            vistgt[5][j]=tgt[16][j];
            n=5;

            }

            }

        goto END;

            }

            }

        if (xs < xmin)

            {

        if ((ys >= ymin) && (ys <= ymax))

            {

/* Sensor is horizontally aligned with target grids and to left of target, therefore it sees
   left side faces */
        for (j=1; j<=4; j++)

            {

            vistgt[1][j]=tgt[2][j];
            vistgt[2][j]=tgt[3][j];
            vistgt[3][j]=tgt[4][j];
            vistgt[4][j]=tgt[14][j];
            n=4;

/* Allows for skewed view of top block side */
        if (ys < yt)

            {

            vistgt[5][j]=tgt[13][j];
            n=5;

            }
```

```
      if (ys > yt)

            {

            vistgt[5][j]=tgt[15][j];
            n=5;

            }

            }

      goto END;

            }

      if (ys < ymin)

            {

/* Sensor is to left and below target, all of left side and lower faces seen */
      for (j=1; j<=4; j++)

            {

            vistgt[1][j]=tgt[1][j];
            vistgt[2][j]=tgt[2][j];
            vistgt[3][j]=tgt[3][j];
            vistgt[4][j]=tgt[4][j];
            vistgt[5][j]=tgt[12][j];
            vistgt[6][j]=tgt[11][j];
            vistgt[7][j]=tgt[13][j];
            vistgt[8][j]=tgt[14][j];
            n=8;

            }

      goto END;

            }

      if (ys > ymax)

            {
```

```
/* Sensor is to left side and above target, all of left side and upper faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[2][j];
                vistgt[2][j]=tgt[3][j];
                vistgt[3][j]=tgt[4][j];
                vistgt[4][j]=tgt[5][j];
                vistgt[5][j]=tgt[5][j];
                vistgt[6][j]=tgt[7][j];
                vistgt[7][j]=tgt[14][j];
                vistgt[8][j]=tgt[15][j];
                n=8;

                }

        goto END;

                }

                }

/* Sensor is right of target */
        if ((ys >= ymin) && (ys <= ymax))

                {

/* Sensor horizontally aligned w/target and right of target, therefore right faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[8][j];
                vistgt[2][j]=tgt[9][j];
                vistgt[3][j]=tgt[10][j];
                vistgt[4][j]=tgt[16][j];
                n=4;

/* Allows a skewed view of a top side */
        if (ys < yt)

                {
```

```
            vistgt[5][j]=tgt[13][j];
            n=5;

            }

    if (ys > yt)

            {

            vistgt[5][j]=tgt[15][j];
            n=5;

            }

            }

    goto END;

            }

    if (ys < ymin)

            {

/* Sensor is right of and below target, all right and lower faces seen */
    for (j=1; j<=4; j++)

            {

            vistgt[1][j]=tgt[8][j];
            vistgt[2][j]=tgt[9][j];
            vistgt[3][j]=tgt[10][j];
            vistgt[4][j]=tgt[1][j];
            vistgt[5][j]=tgt[12][j];
            vistgt[6][j]=tgt[11][j];
            vistgt[7][j]=tgt[13][j];
            vistgt[8][j]=tgt[16][j];
            n=8;

            }

    goto END;

            }
```

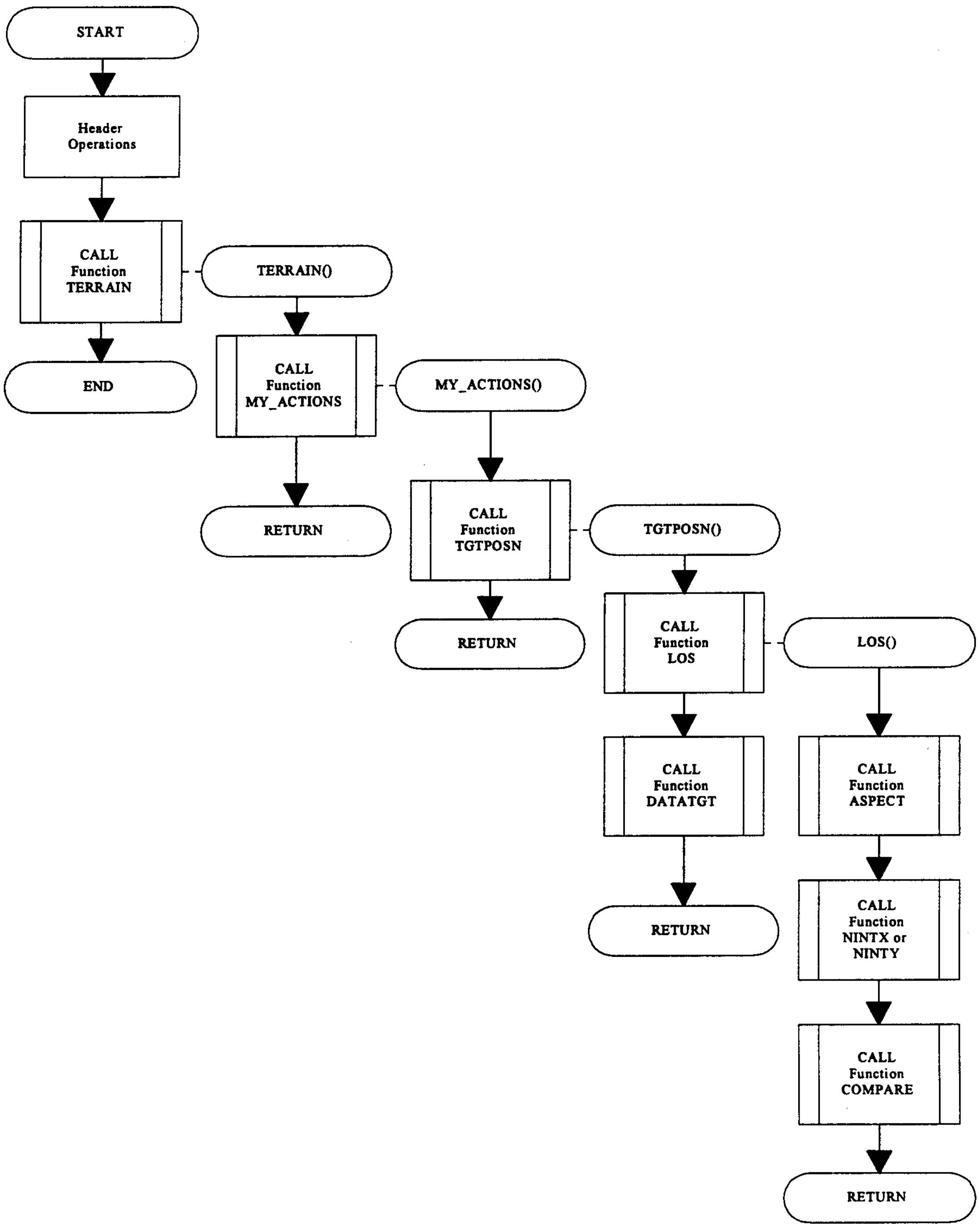
```
        if (ys > ymax)

                {

 /* Sensor is right and above target, all right and upper faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[8][j];
                vistgt[2][j]=tgt[9][j];
                vistgt[3][j]=tgt[10][j];
                vistgt[4][j]=tgt[5][j];
                vistgt[5][j]=tgt[6][j];
                vistgt[6][j]=tgt[7][j];
                vistgt[7][j]=tgt[16][j];
                vistgt[8][j]=tgt[15][j];
                n=8;

                }

        goto END;

                }

        END:

        return;
}

/****************************** nintx() ******************************/
/* This function round up or down to the nearest integer for ix */
nintx()

{

        ix=floor(x);
        zx=fabs(x-ix);

        if (zx >= .5)

                {

                ix=ceil(x);
                }

        else

                {

                ix=floor(x);

                }

        return;
}

/****************************** ninty() *******************************/
/* This function round up or down to the nearest integer for iy */
ninty()

{

        iy=floor(y);
        zy=fabs(y-iy);

        if (zy >= .5)

                {

                iy=ceil(y);

                }
        else
                {

                iy=floor(y);

                }

        return;
}

/***************************** compare() ****************************/
/* This function compares LOS data to terrain data to see if LOS is obstructed */
compare()

{
```

```
/* Compares LOS height to ground height */

        if (z < zdirt)

                {

                nolos=1;

                }

        else if (z < ztree)

                {

/* The following determine attenuation due to vegetation.  If feature is 200 m away
   then appears as solid object, distance arbitrarily selected.  Checks UCI if object has
   height but no UCI assume to be trunk of object/man made structure, LOS blocked */
        if (uci[ix*50+iy] == 0)

                {

                nolos=2;

                }

        else if (z > uci[ix*50+iy])

                {

/* Checks nature bit, structures block LOS. Manmade objects have a nature bit of 0 */
        if (nat[ix*50+iy] == 0)

                {

                nolos=3;

                }

/* If program passes  test above then obstruction is vegetation and any other parts of
   tree will be assumed as foliage.  Current assumption is foliage of 1 meter thickness
   has an attenuation of 30%.  This value is modified as a function of distance until the
   modified value reaches an attenuation of 100% at terminal distance, 200 meters.  At
   200 meters all objects appear solid.  It is assumed the modification factor is linear */
        if (dist > 200)
```

```
                {

                nolos=4;

                }
        else
                {

                attenf=denfol*(1+2.33*dist/200);

                }

                }

/* Allows for sensor hiding behind or in foliage to see through w/out attenuation */
        if (dist <= 1)

                {

                attenf=0;

                }

/* Sums attenuations */
                atten=atten+attenf;

/* If total attenuation exceeds 95%, LOS is blocked */
        if (atten > .95)

                {

                nolos=5;

                }

                }

        return;
}

/**************************** datatgt() ******************************/
/* This function displays to screen:  range, % of target faces visible, and total area of
   target faces visible */
```

```
datatgt()

{

        prcnt=floor((r / n)*100);
        printf("\n%d\t%d\t%3.2f\t%d\t%3.2f\n",cxt,cyt,range,prcnt,totarea);

        return;
}

/****************************** tree() ******************************/
/* This function creates trees & places them on the terrain from info in "tree.dat" */
static void tree()

{

/* Declare variables */
        int row;
        float rad,x,z;

/* Opens file "tree.dat" and reads dimensional data */
        fp1=fopen("tree.dat","r");

        for (row=0; row<7; row++)

        {

        fscanf(fp1," %f %f %f\n",&mdata[row][0],&mdata[row][1],&mdata[row][2]);

/* Defines variables rad, x, z */
        rad=mdata[row][0];x=mdata[row][1];z=mdata[row][2];

/* Defines cube as a new sphere and cyl as a new cylinder, attach's them together */
        factors[X]=factors[Z]=.75;factors[Y]=1.2;
        sphere = WTobject_newsphere(rad,5,5,1,1,1);
        WTobject_stretch(sphere,factors,p,WTFRAME_WORLD);
        cyl = WTobject_newcylinder(rad/1.25,rad/5,4,1,1,1);
        dp4[X]=0;dp4[Y]=-(.95*rad);dp4[Z]=0;
        WTobject_translate(sphere,dp4,WTFRAME_WORLD);
        WTobject_attach(sphere,cyl);
        WTobject_add(sphere);
        dp5[X]=x;dp5[Y]=-rad/4;dp5[Z]=z;
        WTobject_translate(sphere,dp5,WTFRAME_WORLD);
        }

        fclose(fp1);
        return;
}
```

APPENDIX J. FLOWCHART PROG3.C



APPENDIX K. PROGRAM PROG3.C

```
/* PROGRAM PROG3.C */
/*********************************************************************/
/* This program calculates Line Of Sight (LOS) data and visualizes a target moving*/
/* through a 35x50 grid using a database of Pegasus numbers and is designed to    */
/* incorporate attenuation by vegetation.  The main section of this program calls  */
/* function terrain() which is used to start WorldToolKit and create a universe    */
/* where there is a flat terrain with trees and a building located on it.  The trees and*/
/* building are represented as the LOS algorithm would see them.  The simulation  */
/* has the target moving along a path behind the trees and building.  Function     * /
/* terrain() calls function los() which is used to calculate target data.          */
/*********************************************************************/
/* Standard C and WorldToolKit header files */
#include <stdio.h>
#include <math.h>
#include"wt.h"
#include"wt.p"

/* Defines a pointer spaceball to structure type WTsensor */
static WTsensor *spaceball=NULL;

/* Defines windows, viewpoints, nodes, and paths */
static WTwindow *win1, *win2, *win3;
static WTviewpoint *view, *objview1, *objview2, *objview3;

/* Defines objects */
WTobject *bldg, *copy, *cyl, *original, *sensor, *sphere, *cube, *terrobj;
WTobject *nodeobj, *tree3m1, *tree3m2, *tree3m3, *tree3m4, *tree3m5;
WTobject *tree8m1c, *tree8m1o1, *tree8m1o2, *tree8m1o3, *tree8m1o4;
WTobject *tree8m2c, *tree8m2o1, *tree8m2o2, *tree8m2o3, *tree8m2o4;
WTobject *tree, *target;

/* Defines path */
static WTpath *tnkpth;

/* Defines pointers to WTp3, WTq,and WTpq structures */
WTp3 at,dp1,dp2,dp3,dp4,dp5,dp6,dp7,dp8,dir,factors,p,p1,p2,ppp1,ppp2,pp,pt;
WTp3 dp9,dp10,dp11,dp12,dp13,dp14,dp15,dp16,dp17,dp18,dp19,dp20;
WTp3 at1, dir1;
WTq pp1,pp2;
WTpq modelview;
```

```
/* Defines light object to be a pointer to type WTlight */
WTlight *mylight;

/* Initialize arrays and declares variables */
int ans,c,cxt,cyt,ho,ht,i,ix,iy,istart,istop,j,n,nmbr,nolos,prcnt;
int win,wdow,xs,xt,xmax,xmin,xstart,ys,yt,ymax,ymin,ystart,zs,zt,zdirt,ztree;
int binelev[1750],binnat[1750],binuci[1750],binvghindex[1750],binvid[1750];
int elev[1750],nat[1750],realvgh[10],tgt[17][5],tile[1750];
int uci[1750],vgh[1750],vid[1750],vghindex[1750],vistgt[9][9];
float aproj,atten,attenf,dx,dy,dz,denfol,dist,r,rad,rdx,rdy,row,rdns,range;
float totarea,visaproj,x,y,y_rotate,z,zx,zy;
float mdata[7][3],tc;

/* Standard C file pointers */
FILE *fp;
FILE *fp1;

main()

{

/* Reads database information into array from data file "btlfld_terr.dat" */
        fp=fopen("btlfld_terr.dat","r");
        for (i=0; i<=1749; i++)

                {

                fscanf(fp," %d", &tile[i]);

                }

        fclose(fp);

/* Assigns 1 meter of foliage an attenuation of 30% of the LOS */
        denfol=0.3;

/* The vegetation height from pegasus has values of 0-15, each of these values
   represents a particular height or range of heights. The following is used to correlate
   the vgh value to a meaningful height */
        realvgh[0]=0;
        realvgh[1]=0;
        realvgh[2]=1;
        realvgh[3]=2;
        realvgh[4]=3;
        realvgh[5]=4;
        realvgh[6]=5;
        realvgh[7]=8;
        realvgh[8]=10;
        realvgh[9]=15;
        realvgh[10]=20;
        realvgh[11]=25;
        realvgh[12]=30;
        realvgh[13]=35;
        realvgh[14]=40;
        realvgh[15]=47;

/* Once the database for the grid desired has been read into an array, the components
   of information are extracted from the 32 bit number using the AND and SHIFT
   functions. The groups of information of use in the program are placed in their own
   arrays for rapid recall during calculations as follows:
   elevation of highest point in grid        - ELEV
   under cover index                         - UCI
   converted vegetation height               - VGH
   vegetation ID                             - VID
   nature bit                                - NAT */
      for (i=0; i<=1749; i++)

            {

            binuci[i]=tile[i] & 786432;
            binvghindex[i]=tile[i] & 15360;
            binvid[i]=tile[i] & 768;
            binnat[i]=tile[i] & 128;
            binelev[i]=tile[i] & 4292870144;

            uci[i] = binuci[i] >> 18;
            vghindex[i] = binvghindex[i] >> 10;
            vgh[i]=realvgh[vghindex[i]];
            vid[i] = binvid[i] >> 8;
            nat[i] = binnat[i] >> 7;
            elev[i] = binelev[i] >> 21;

            }

/* This program is currently written to run in a stand alone mode.  Sensor information
       has a default value, but can be changed by answering prompts from the keyboard
   Prompts to accept default values or input sensor location and height */
      printf("\nThe Default Sensor Coordinates are xs = 0, zs = 0");
```

```
        printf("\nand hs = 1\n");
        printf("\nDo you wish to Change these values?  1=Yes, 2=No\n");
        scanf(" %d",&ans);

        if (ans == 1)
                {

                BEGIN:

                printf("\nEnter Sensor X Coordinate (X,Y)\n");
                scanf("%d,%d",&xs,&ys);
                printf("\nEnter Sensor Height\n");
                scanf("%d",&ho);
                printf("\nAre You Sure?  1=Yes  2=No \n");
                scanf(" %d", &ans);

        if (ans == 2)

                {

                goto BEGIN;

                }

                }
        else
                {

                xs=0;
                ys=0;
                ho=1;

                }

        printf("\nPress Spaceball Button 1 to add a window\n");
        printf("\nPress Spaceball Button 2 to delete window added\n");
        printf("\nPress Spaceball Button 3 to start target motion\n");
        printf("\nPress Spaceball Button 4 to replay simulation\n");
        printf("\nPress Spaceball Button 8 to end simulation\n");

/* Calls function terrain() */
        terrain();

        return;
```

```
}

/****************************** terrain() ******************************/
/* This function is WorldToolKit */
terrain()

{
        void my_actions();

/* Creates new universe */
        WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

/* Creates the object terrobj */
        terrobj=WTterrain_flat(0.0,35,50,0,0,1.0,1.0,0x0f0,0x0e0,1.0);

/* Defines pose of lights, and create light object*/
        at[0]=0.0;at[1]=-25.0;at[2]=25.0;
        dir[0]=.5;dir[1]=.5;dir[2]=.5;
        mylight=WTlight_new(at,dir,1.0);
        at1[0]=35.0;at1[1]=-25.0;at1[2]=25.0;
        dir1[0]=-.5;dir1[1]=.5;dir1[2]=-.5;
        mylight=WTlight_new(at1,dir1,1.0);

/* Creates the object spaceball or mouse */
        spaceball=WTspaceball_new(COM1);

/* Creates small building, sensor, and target, and places them on the terrain */
        bldg=WTobject_newblock(3,-8,3,1,1);
        dp1[X]=29;dp1[Y]=-4;dp1[Z]=29;
        WTobject_translate(bldg,dp1,WTFRAME_WORLD);
        WTobject_changecolor(bldg,0xfff,0x777);

        sensor=WTobject_newblock(.5,.5,.5,1,1);
        dp2[X]=0.0;dp2[Y]=-.250;dp2[Z]=0.0;
        WTobject_setposition(sensor,dp2);
        WTobject_changecolor(sensor,0xfff,0xf0f);

/* Target is loaded from external Autodesk dxf file */
        cube=WTobject_new("cubetgt.dxf",&modelview,1.0,FALSE,FALSE);
        dp3[X]=15.0;dp3[Y]=5.0;dp3[Z]=25.0;
        WTobject_setposition(cube,dp3);
        WTobject_changecolor(cube,0x000,0xfff);
```

```
/* Target path is loaded from external file */
        nodeobj=WTobject_newblock(.1,.1,.1,1,1);
        tnkpth=WTpath_load("cube.pth",nodeobj);

/* Creates trees */
        tc = 0x777;
        tree3m1=WTobject_newblock(1,3,1,1,1);
        dp4[X]=17.0;dp4[Y]=-1.5;dp4[Z]=27.0;
        WTobject_setposition(tree3m1,dp4);
        WTobject_changecolor(tree3m1,0xfff,tc);
        tree3m2=WTobject_newblock(1,3,1,1,1);
        dp5[X]=17.0;dp5[Y]=-1.5;dp5[Z]=33.0;
        WTobject_setposition(tree3m2,dp5);
        WTobject_changecolor(tree3m2,0xfff,tc);
        tree3m3=WTobject_newblock(1,3,1,1,1);
        dp6[X]=19.0;dp6[Y]=-1.5;dp6[Z]=31.0;
        WTobject_setposition(tree3m3,dp6);
        WTobject_changecolor(tree3m3,0xfff,tc);
        tree3m4=WTobject_newblock(1,3,1,1,1);
        dp7[X]=18.0;dp7[Y]=-1.5;dp7[Z]=34.0;
        WTobject_setposition(tree3m4,dp7);
        WTobject_changecolor(tree3m4,0xfff,tc);
        tree3m5=WTobject_newblock(1,3,1,1,1);
        dp8[X]=22.0;dp8[Y]=-1.5;dp8[Z]=33.0;
        WTobject_setposition(tree3m5,dp8);
        WTobject_changecolor(tree3m5,0xfff,tc);
        tree8m1c=WTobject_newblock(1,8,1,1,1);
        dp9[X]=12.0;dp9[Y]=-4.0;dp9[Z]=36.0;
        WTobject_setposition(tree8m1c,dp9);
        WTobject_changecolor(tree8m1c,0xfff,tc);
        tree8m1o1=WTobject_newblock(1,4,1,1,1);
        dp10[X]=12.0;dp10[Y]=-3.0;dp10[Z]=35.0;
        WTobject_setposition(tree8m1o1,dp10);
        WTobject_changecolor(tree8m1o1,0xfff,tc);
        tree8m1o2=WTobject_newblock(1,4,1,1,1);
        dp11[X]=13.0;dp11[Y]=-3.0;dp11[Z]=36.0;
        WTobject_setposition(tree8m1o2,dp11);
        WTobject_changecolor(tree8m1o2,0xfff,tc);
        tree8m1o3=WTobject_newblock(1,4,1,1,1);
        dp12[X]=12.0;dp12[Y]=-3.0;dp12[Z]=37.0;
        WTobject_setposition(tree8m1o3,dp12);
        WTobject_changecolor(tree8m1o3,0xfff,tc);
        tree8m1o4=WTobject_newblock(1,4,1,1,1);
        dp13[X]=11.0;dp13[Y]=-3.0;dp13[Z]=36.0;
        WTobject_setposition(tree8m1o4,dp13);
        WTobject_changecolor(tree8m1o4,0xfff,tc);
        tree8m2c=WTobject_newblock(1,8,1,1,1);
        dp14[X]=26.0;dp14[Y]=-4.0;dp14[Z]=35.0;
        WTobject_setposition(tree8m2c,dp14);
        WTobject_changecolor(tree8m2c,0xfff,tc);
        tree8m2o1=WTobject_newblock(1,4,1,1,1);
        dp15[X]=26.0;dp15[Y]=-3.0;dp15[Z]=34.0;
        WTobject_setposition(tree8m2o1,dp15);
        WTobject_changecolor(tree8m2o1,0xfff,tc);
        tree8m2o2=WTobject_newblock(1,4,1,1,1);
        dp16[X]=27.0;dp16[Y]=-3.0;dp16[Z]=35.0;
        WTobject_setposition(tree8m2o2,dp16);
        WTobject_changecolor(tree8m2o2,0xfff,tc);
        tree8m2o3=WTobject_newblock(1,4,1,1,1);
        dp17[X]=26.0;dp17[Y]=-3.0;dp17[Z]=36.0;
        WTobject_setposition(tree8m2o3,dp17);
        WTobject_changecolor(tree8m2o3,0xfff,tc);
        tree8m2o4=WTobject_newblock(1,4,1,1,1);
        dp18[X]=25.0;dp18[Y]=-3.0;dp18[Z]=35.0;
        WTobject_setposition(tree8m2o4,dp18);
        WTobject_changecolor(tree8m2o4,0xfff,tc);

/* Prints to screen title headings */
        printf("\nxt\tyt\trange\tprcnt\ttotarea");
        printf("\n___\t___\t______\t______\t_______\n");

/* Sets universe action function to my_actions() */
        WTuniverse_setactions(my_actions);

/* Prepares for simulation to start */

        WTuniverse_ready();

/* Scales spaceball or mouse sensitivity to the size of the universe */
        WTsensor_setsensitivity(spaceball, 5.0 *WTuniverse_getradius());

/* Connects the viewpoint to the spaceball or mouse */
        WTviewpoint_addsensor(view,spaceball);

/* Puts some lights on in the universe */
        WTlight_setambient(0.4);
```

```
/* Enters main simulation loop and starts simulation */
        WTuniverse_go();

/* Deletes universe, simulation loop must be exited to reach this statement */
        WTuniverse_delete();

        return;
}

/************************** my_actions() ****************************/
/* This function determines what actions will occur in the universe */
void my_actions()

{

/* Sets default universe viewpoint */
        view=WTuniverse_getviewpoint();
        p1[X]=xs;p1[Y]=-ho;p1[Z]=ys;
        pp1[X]=0.0;pp1[Y]=0.0;pp1[Z]=0.0;pp1[W]=1.0;
        ppp1[X]=1.0;ppp1[Y]=0.0;ppp1[Z]=1.3;
        WTviewpoint_setposition(view,p1);
        WTviewpoint_setorientation(view,pp1);
        WTviewpoint_setdirection(view,ppp1);

/* Pressing spaceball button ONE adds an additional window */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON1)

        {

        win2=WTwindow_new(505,500,250,400,WTWINDOW_DEFAULT);
        objview2=WTviewpoint_new();

        p2[X]=17.5;p2[Y]=-25.0;p2[Z]=25.0;
        pp2[X]=0.0;pp2[Y]=0.0;pp2[Z]=0.0;pp2[W]=1.0;
        ppp2[X]=0.0;ppp2[Y]=1.0;ppp2[Z]=0.0;
        WTviewpoint_setposition(objview2,p2);
        WTviewpoint_setorientation(objview2,pp2);
        WTviewpoint_setdirection(objview2,ppp2);
        WTwindow_setviewpoint(win2,objview2);

        }

/* Pressing spaceball button TWO deletes the first window added */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON2)
        {

        WTwindow_delete(win2);

        }

/* Pressing spaceball button THREE starts the target motion */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON3)

        {

        WTpath_setobject(tnkpth,cube);
        WTpath_play(tnkpth);
        WTpath_record(tnkpth);

        }

/* Pressing spaceball button FOUR restarts the target motion */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON4)

        {

        WTpath_stop(tnkpth);
        WTpath_rewind(tnkpth);
        WTpath_play(tnkpth);

        }

/* Based on target position, function tgtposn() called */
        WTobject_getposition(cube,pp);

        if (pp[X] == 2)

        {

        tgtposn();

        }

        if (pp[X] == 7)

        {

        tgtposn();
}
if (pp[X] == 12)
{
tgtposn();
}
if (pp[X] == 17)
{
tgtposn();
}
if (pp[X] == 22)
{
tgtposn();
}
if (pp[X] == 26)
{
tgtposn();
}
if (pp[X] == 30)
{
tgtposn();
}
if ((pp[X] == 33) && (pp[Z] == 37))
```

```
        {
        tgtposn();
        }
        if ((pp[X] == 33) && (pp[Z] == 32))
        {
        tgtposn();
        }
        if ((pp[X] == 33) && (pp[Z] == 27))
        {
        tgtposn();
        }
        if ((pp[X] == 33) && (pp[Z] == 22))
        {
        tgtposn();
        }
/* Pressing spaceball button EIGHT terminates simulation */
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON8)
        {
        WTuniverse_stop();
        }
        return;
}
```

```
/**************************** tgtposn() ******************************/
/* This function calls los() and datatgt(), receives value of xt and zt from
   WTobject_getposition, and displays target data to screen */
tgtposn()

{

        xt=pp[X];yt=pp[Z];

/* Calls function los() to determine LOS to target faces, and other calculations */
        los();

/* Calls function datatgt() to display target data */
        datatgt();

/* Zeros out total area and percent data */
        totarea=0;
        prcnt=0.0;

        return;
}

/******************************** los() ********************************/
/* This function calculates LOS data */
los()

{

/* Begins loop inputting targets moving locations */
                cxt=xt;
                cyt=yt;
                range=sqrt((float)pow((xt-xs),2)+(float)pow((yt-ys),2));

/* Calls the function aspect() to determine how much of the target is presented for
   possible LOS.  Vistgt[ ][ ] is the array returned holding the grid locatiõn and
   information on the faces of the target which may be seen, n = the number of
   possible detections and is used for looping the algorithm */
                aspect();

/* Loops to check possible LOS for all surfaces presented by target */
                r=0;

        for (j=1; j<=n; j++)
```

```
                {

/* Zeros atten for each run and gets target grid height data for stepping LOS */
                atten=0;
                attenf=0;
                xt=vistgt[j][1];
                yt=vistgt[j][2];
                ht=vistgt[j][3];

/* Calculates target and observer heights. First ground height must be found by
   subtracting vegetation height from absolute height.  Then sensor and target heights
   above ground are added to obtain absolute elevations of sensor and target */
                zs=elev[xs*50+ys]-vgh[xs*50+ys];
                zt=elev[xt*50+yt]-vgh[xt*50+yt];
                zs=zs+ho;
                zt=zt+ht;

/* Determines difference btwn x & y coordinates, and convert from integer to real */
                rdx=(xt-xs);
                rdy=(yt-ys);

/* If rdy > rdx, skip to stepping in y direction, else proceed stepping in x direction */
        if (rdy >= rdx)

                {

                goto YSTEP;

                }
        else
                {

                y=ys;
                dy=(yt-ys)/rdx;
                z=zs;
                dz=(zt-zs)/rdx;

/* This if-else statement ensures moving from sensor to target.  Move only from
   sensor because dist from sensor will affect attenuation level of any obstructions */
        if (xt > xs)

                {
```

```
/* Apply slope to each step to determine grid passing thru and LOS height in that grid,
   compare height to ground height, no LOS will exist if ground height > LOS height:
   z           = height of LOS
   ztree       = height of vegetation
   zdirt       = height of the ground */
               istart=xs+1;
               istop=xt;
               ystart=y;

       for (ix=istart; ix<=istop; ix++)

               {

               y=y+dy;
               z=z+dz;
               ninty();
               ztree=elev[ix*50+iy];
               zdirt=ztree-vgh[ix*50+iy];

/* Calculate dist from sensor to grid where LOS currently in heading toward target */
               dist=sqrt((float)pow((istart-ix),2)+(float)pow((ystart-iy),2));

/* Calls function compare() */
       compare();

       if (nolos != 0)

               {

               goto FINISH;

               }

               }

      goto CONTINUE;

              }
      else
              {

              istart=xs-1;
              istop=xt;
              ystart=y;
```

```
     for (ix=istart; ix<=istop; ix--)

             {

             y=y+dy;
             z=z+dz;
             ninty();
             ztree=elev[ix*50+iy];
             zdirt=ztree-vgh[ix*50+iy];
             dist=sqrt((float)pow((istart-ix),2)+(float)pow((ystart-iy),2));

/* Calls function compare() */
     compare();

     if (nolos != 0)

             {

             goto FINISH;

             }

             }

     goto CONTINUE;

             }

             }

/* This section is used if stepping in y direction and same as stepping in x direction */
     YSTEP:

             x=xs;
             dx=(xt-xs)/rdy;
             z=zs;
             dz=(zt-zs)/rdy;

     if (yt > ys)

             {

             istart=ys+1;
             istop=yt;
```

```
                xstart=x;

          for (iy=istart; iy<=istop; iy++)

                {

                x=x+dx;
                z=z+dz;
                nintx();
                ztree=elev[ix*50+iy];
                zdirt=ztree-vgh[ix*50+iy];
                dist=sqrt((float)pow((xstart-ix),2)+(float)pow((istart-iy),2));

/* Calls function compare() */
          compare();

          if (nolos != 0)

                {

                goto FINISH;

                }

                }

          goto CONTINUE;

                }
          else
                {

                istart=ys-1;
                istop=yt;
                xstart=x;

          for (iy=istart; iy<=istop; iy--)

                {

                x=x+dx;
                z=z+dz;
                nintx();
                ztree=elev[ix*50+iy];
```

```
                    zdirt=ztree-vgh[ix*50+iy];
                    dist=sqrt((float)pow((xstart-ix),2)+(float)pow((istart-iy),2));

 /* Calls function compare() */
           compare();

           if (nolos != 0)

                    {

                    goto FINISH;

                    }

                    }

           goto CONTINUE;

                    }

           CONTINUE:

/* At this point LOS has atten = 0 or a number.  Apply this attenuation factor to proj
   area of target to reduce area of visibility.  Proj area is determined based on surface
   direction of face in question.  This value is 1 for a vertical surface and 2 for a
   horizontal surface.  This value is stored in vistgt[ ][ ] array which describes target */
          if (vistgt[j][4] == 0)

                    {

                    aproj=rdy/sqrt(pow(rdy,2)+pow(rdx,2));

                    }
          else
                    {

                    aproj=rdx/sqrt(pow(rdy,2)+pow(rdx,2));

                    }

/* Sees if there is any attenuation to be applied, if so apply it */
          if (atten == 0)

                    {
```

```
                visaproj=aproj;

                }
        else
                {

                visaproj=(1-atten)*aproj;

                }

 /* Sums total visible area presented by target */
                totarea=totarea+visaproj;

 /* Zeros aproj and visaproj for next LOS */
                aproj=0;
                visaproj=0;
                r=r+1;

        FINISH:

                nolos=0;

                }

                return;
}

/****************************** aspect() ******************************/
/* This subroutine assigns target grid locations based on central input location.  It then
   uses sensor and target locations to determine which faces of target are ideally visible
   to sensor.  Faces which are visible have there information stored in array vistgt[ ][ ]
   for return to main program.  The number of faces ideally visible are also returned.
   Assign target data to array tgt[ ][ ] based on center grid of target 3x3 each exterior
   vertical face of target is represented */
aspect()

{

        tgt[1][1]=xt-1;
        tgt[1][2]=yt-2;
        tgt[1][4]=0;
        tgt[2][1]=xt-2;
        tgt[2][2]=yt-1;
        tgt[2][4]=1;
```

```
tgt[3][1]=xt-2;
tgt[3][2]=yt;
tgt[3][4]=1;
tgt[4][1]=xt-2;
tgt[4][2]=yt+1;
tgt[4][4]=1;
tgt[5][1]=xt-1;
tgt[5][2]=yt+2;
tgt[5][4]=0;
tgt[6][1]=xt;
tgt[6][2]=yt+2;
tgt[6][4]=0;
tgt[7][1]=xt+1;
tgt[7][2]=yt+2;
tgt[7][4]=0;
tgt[8][1]=xt+2;
tgt[8][2]=yt+1;
tgt[8][4]=1;
tgt[9][1]=xt+2;
tgt[9][2]=yt;
tgt[9][4]=1;
tgt[10][1]=xt+2;
tgt[10][2]=yt-1;
tgt[10][4]=1;
tgt[11][1]=xt+1;
tgt[11][2]=yt-2;
tgt[11][4]=0;
tgt[12][1]=xt;
tgt[12][2]=yt-2;
tgt[12][4]=0;
tgt[13][1]=xt;
tgt[13][2]=yt-1;
tgt[13][4]=0;
tgt[14][1]=xt-1;
tgt[14][2]=yt;
tgt[14][4]=1;
tgt[15][1]=xt;
tgt[15][2]=yt+1;
tgt[15][4]=0;
tgt[16][1]=xt+1;
tgt[16][2]=yt;
tgt[16][4]=1;
```

```
/* Assigns target heights, in tgt[ ][ ] */
        for (j=1; j<=12; j++)

                {

                tgt[j][3]=1;

                }

        for (j=13; j<=16; j++)

                {

                tgt[j][3]=2;

                }

/* Establishes bounds of the target */
                xmax=xt+1;
                xmin=xt-1;
                ymax=yt+1;
                ymin=yt-1;

/* Determines visible sectors of target */
        if ((xs <= xmax) && (xs >= xmin))

                {

        if (ys > ymax)

                {

/* Sensor is directly above target grids, upper faces */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[5][j];
                vistgt[2][j]=tgt[6][j];
                vistgt[3][j]=tgt[7][j];
                vistgt[4][j]=tgt[15][j];
                n=4;
```

```
/* Allows for skewed view of a top block side */
        if (xs < xt)

                {

                vistgt[5][j]=tgt[14][j];
                n=5;

                }

        if (xs > xt)

                {

                vistgt[5][j]=tgt[16][j];
                n=5;

                }

                }

        goto END;

                }
        else
                {

/* If the sensor is vertically in line with target grids and it is not above target it must
   be below it, therefore it sees lower faces */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[1][j];
                vistgt[2][j]=tgt[12][j];
                vistgt[3][j]=tgt[11][j];
                vistgt[4][j]=tgt[13][j];
                n=4;

/* Allows for skewed view of a top block side */
        if (xs < xt)

                {
```

```
                vistgt[5][j]=tgt[14][j];
                n=5;

                }

        if (xs > xt)

                {

                vistgt[5][j]=tgt[16][j];
                n=5;

                }

                }

        goto END;

                }

                }

        if (xs < xmin)

                {

        if ((ys >= ymin) && (ys <= ymax))

                {

/* Sensor is horizontally aligned with target grids and to left of target, therefore it sees
   left side faces */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[2][j];
                vistgt[2][j]=tgt[3][j];
                vistgt[3][j]=tgt[4][j];
                vistgt[4][j]=tgt[14][j];
                n=4;

/* Allows for skewed view of top block side */
        if (ys < yt)
```

```
                {

                vistgt[5][j]=tgt[13][j];
                n=5;

                }

        if (ys > yt)

                {

                vistgt[5][j]=tgt[15][j];
                n=5;

                }

                }

        goto END;

                }

        if (ys < ymin)

                {

/* Sensor is to left and below target, all of left side and lower faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[1][j];
                vistgt[2][j]=tgt[2][j];
                vistgt[3][j]=tgt[3][j];
                vistgt[4][j]=tgt[4][j];
                vistgt[5][j]=tgt[12][j];
                vistgt[6][j]=tgt[11][j];
                vistgt[7][j]=tgt[13][j];
                vistgt[8][j]=tgt[14][j];
                n=8;

                }

        goto END;
```

```
                }

        if (ys > ymax)

                {

 /* Sensor is to left side and above target, all of left side and upper faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[2][j];
                vistgt[2][j]=tgt[3][j];
                vistgt[3][j]=tgt[4][j];
                vistgt[4][j]=tgt[5][j];
                vistgt[5][j]=tgt[5][j];
                vistgt[6][j]=tgt[7][j];
                vistgt[7][j]=tgt[14][j];
                vistgt[8][j]=tgt[15][j];
                n=8;

                }

        goto END;

                }

                }

/* Sensor is right of target */
        if ((ys >= ymin) && (ys <= ymax))

                {

/* Sensor horizontally aligned w/target and right of target, therefore right faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[8][j];
                vistgt[2][j]=tgt[9][j];
                vistgt[3][j]=tgt[10][j];
                vistgt[4][j]=tgt[16][j];
                n=4;
```

```
/* Allows a skewed view of a top side */
        if (ys < yt)

                {

                vistgt[5][j]=tgt[13][j];
                n=5;

                }

        if (ys > yt)

                {

                vistgt[5][j]=tgt[15][j];
                n=5;

                }

                }

        goto END;

                }

        if (ys < ymin)

                {

/* Sensor is right of and below target, all right and lower faces seen */
        for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[8][j];
                vistgt[2][j]=tgt[9][j];
                vistgt[3][j]=tgt[10][j];
                vistgt[4][j]=tgt[1][j];
                vistgt[5][j]=tgt[12][j];
                vistgt[6][j]=tgt[11][j];
                vistgt[7][j]=tgt[13][j];
                vistgt[8][j]=tgt[16][j];
                n=8;
                }

         goto END;

                }

         if (ys > ymax)

                {

  /* Sensor is right and above target, all right and upper faces seen */
         for (j=1; j<=4; j++)

                {

                vistgt[1][j]=tgt[8][j];
                vistgt[2][j]=tgt[9][j];
                vistgt[3][j]=tgt[10][j];
                vistgt[4][j]=tgt[5][j];
                vistgt[5][j]=tgt[6][j];
                vistgt[6][j]=tgt[7][j];
                vistgt[7][j]=tgt[16][j];
                vistgt[8][j]=tgt[15][j];
                n=8;

                }

        goto END;

                }

        END:

        return;
}

/******************************* nintx() *******************************/
/* This function round up or down to the nearest integer for ix */
nintx()

{

        ix=floor(x);
        zx=fabs(x-ix);
```

```
         if (zx >= .5)

                  {

                  ix=ceil(x);

                  }
         else
                  {

                  ix=floor(x);

                  }

         return;
}

/****************************** ninty() ******************************/
/* This function round up or down to the nearest integer for iy */
ninty()

{

         iy=floor(y);
         zy=fabs(y-iy);

         if (zy >= .5)

                  {

                  iy=ceil(y);

                  }
         else
                  {

                  iy=floor(y);

                  }

         return;
}
```

```
/**************************** compare() ****************************/
/* This function compares LOS data to terrain data to see if LOS is obstructed */
compare()

{

/* Compares LOS height to ground height */
        if (z < zdirt)

                {

                nolos=1;

                }

        else if (z < ztree)

                {

/* The following determine attenuation due to vegetation.  If feature is 200 m away
   then appears as solid object, distance arbitrarily selected.  Checks UCI, if object has
   height but no UCI assume to be trunk of object/manmade structure, LOS blocked */
        if (uci[ix*50+iy] == 0)

                {

                nolos=2;

                }

        else if (z > uci[ix*50+iy])

                {

/* Checks nature bit, structures block LOS. Manmade objects have a nature bit of 0 */
        if (nat[ix*50+iy] == 0)

                {

                nolos=3;

                }
/* If program passes above test then obstruction is vegetation and any other parts of
   tree will be assumed as foliage.  Current assumption is  foliage of 1 meter thickness
   has an attenuation of 30%.  This value is modified as a function of distance until the
   modified value reaches an attenuation of 100% at terminal distance, 200 meters.  At
   200 meters all objects appear solid.  It is assumed the modification factor is linear */
        if (dist > 200)

                {

                nolos=4;

                }
        else
                {

                attenf=denfol*(1+2.33*dist/200);

                }

                }

/* Allows for sensor hiding behind or in foliage to see through w/out attenuation */
        if (dist <= 1)

                {

                attenf=0;

                }

/* Sums attenuations */
                atten=atten+attenf;

/* If total attenuation exceeds 95%, LOS is blocked */
        if (atten > .95)

                {

                nolos=5;

                }

                }
        return;
}

/**************************** datatgt() ******************************/
/* This function displays to screen:  range, % of target faces visible, and total area of
   target faces visible */
datatgt()

{


        prcnt=floor((r / n)*100);
        printf("\n%d\t%d\t%3.2f\t%d\t%3.2f\n",cxt,cyt,range,prcnt,totarea);

        return;
}
```

LIST OF REFERENCES

1. TITAN Tactical Applications, *Software Design Manual Janus (A) 2.1 Model*, Contract No. DABT 60-90-D-0002, Delivery Order #37.

2. Dau, Frederick W., *Improving Detection and Acquisition in Janus (A) using Pegasus Database*, Masters Thesis, Department of Mechanical Engineering, Naval Postgraduate School, Monterey, California, March 1994.

3. TITAN Tactical Applications, *Software Programmers Manual Janus (A) 2.1 Model*, Contract No. DABT 60-90-D-0002, Delivery Order #37.

4. U.S. Army TRADOC Analysis Command, *Model Documentation Janus (A) Basic User's Tutorial*, Monterey, California.

5. TITAN Tactical Applications, *User Manual Janus 3.0 Model*, Contract No. DABT 60-90-D-0002, Delivery Order #37.

6. Baer, W., and Akin, J.R., "An Approach for Real-time Database Creation from Aerial Imagery", Nascent Systems Development Inc., Carmel Valley, California.

7. Perry, Greg, *C by Example*, Que Corporation, Carmel, Indiana, 1993.

8. Purdum, Jack, *Guide to C Programming*, Ziff-Davis Press, Emeryville, California, 1992.

9. Sense8 Corporation, *WorkdToolKit Version 2.0 Reference Manual*, Sausalito, California, 1993.

10. Young, John M., *Synthetic Environments for C3 Operations*, Masters Thesis, Department of Mechanical Engineering, Naval Postgraduate School, Monterey, California, September 1994.

11. U.S. Department of the Interior U.S. Geological Survey, *Digital Elevation Models Data Users Guide 5*, Reston, Virginia, 1993.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3.	Department Chairman, Code ME Department of Mechanical Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor Morris R. Driels, Code ME/Dr Department of Mechanical Engineering Naval Postgraduate School Monterey, California 93943-5000	2
5.	Naval Engineering Curricular Officer, Code 34 Department of Mechanical Engineering Naval Postgraduate School Monterey, California 93943-5000	1
6.	Director, TRAC-Monterey P.O. Box 8692 Naval Postgraduate School Monterey, California 93943-0692	2
7.	Professor Wolfgang Baer, Code CS/Ba Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
8.	Adjunct Professor Judith Lind, Code OR/Li Department of Operations Research Naval Postgraduate School Monterey, California 93943-5000	1

9.	Adjunct Professor Bard Mansager, Code MA/Ma Department of Mathematics Naval Postgraduate School Monterey, California 93943-5000	1
10.	Associate Professor Paul Moose, Code CC Command, Control & Communications (C3) Academic Group Naval Postgraduate School Monterey, California 93943-5000	1
11.	Mark R. Whitney 3625 Elkton Drive Chesapeake, Virginia 23321	2